Trabajo Fin de Grado

Manipulación robótica colaborativa mediante el Sistema Operativo Robótico

Robotic manipulation in the Robotic Operating System

Autor:

Inazio Gracia Ruiz

Directores:

Gonzalo López Nicolás
María del Rosario Aragües Muñoz

ESCUELA DE INGENIERÍA Y ARQUITECTURA
2019

AGRADECIMIENTOS

Me gustaría agradecer en primer lugar a mis padres por su paciencia, su apoyo y los consejos que me van dando todos los días. También quiero agradecer a mis dos tutores por brindarme la oportunidad de investigar y aprender en el ámbito de la robótica, por el buen trato recibido y porque sin sus conocimientos este proyecto no se hubiera podido llevar a cabo. Por último y no menos importante, dar las gracias a mi hermano, Rubén y Sergio por ser capaces de animarme cuando las cosas no iban bien.

RESUMEN

MANIPULACIÓN ROBÓTICA COLABORATIVA MEDIANTE El SISTEMA OPERATIVO ROBÓTICO

En la actualidad, hay una clara tendencia a que los robots manipuladores y colaborativos vayan ganando importancia y utilidad en el mundo de la industria. Esto se debe a la persistencia de industrializar todo tipo de trabajos manuales, con el principal fin de ahorrar tiempo y dinero, así como mejorar las condiciones laborables y aumentar la productividad. Es por ello, que saber programar diferentes tipos de robots industriales nos proporciona una gran versatilidad en este contexto. Por dicho motivo, nos centramos en el Sistema ROS (Robotic Operating System), ya que es capaz de trabajar en un mismo entorno con robots fabricados por diferentes empresas.

Cualquier usuario partiendo de un conocimiento nulo acerca de ROS, tras la lectura de este Trabajo Fin de Grado, y con las herramientas adecuadas (un ordenador con conexión a Internet), será capaz de desarrollar una tarea de manipulación en un entorno de simulación. En otras palabras, se describen las pautas a modo de guía para llevar a cabo la implementación en una simulación de tareas de manipulación.

La memoria de este Trabajo comienza por una breve introducción al mundo de la robótica, en especial citando los diferentes tipos de robots que se pueden encontrar, y describiendo con más detalle los robots manipuladores y colaborativos. A continuación, se presenta la guía de ROS en la que se explican las pautas generales sobre qué es, y cómo trabajar con el Sistema Operativo Robótico. Estas explicaciones cerrarán la parte teórica del Trabajo y serán de utilidad para la siguiente parte.

La parte práctica comienza por instalación de ROS. También se definen varias configuraciones útiles del entorno de ROS. Es importante que tanto la instalación como la configuración sean correctas ya que si no pueden aparecer diversos problemas. Es por ello, por lo que se explica cómo realizar diferentes pruebas para comprobar el correcto funcionamiento de ROS.

Las primeras simulaciones se realizan con un robot ABB modelo IRB 120 y sirven de explicación sobre cómo realizar una planificación de movimientos en un entorno de simulación. En ellas se describen los diferentes métodos para mover un manipulador robótico en un escenario. Para ampliar los conocimientos, y como normalmente en las industrias trabajan varios robots simultáneamente en células de trabajo, se ha aplicado uno de los métodos de planificación de movimientos a un caso multirobot.

Finalmente, se procede al desarrollo de dos tareas de manipulación con robots colaborativos de Universal Robots, más concretamente el modelo UR5. Cada tarea viene definida en un escenario diferente y se realiza con dos herramientas distintas, una pinza robótica y una herramienta de vacío.

SUMARIO

1. INTRODUCCIÓN 1

1.1 Motivación y objetivos 1

1.2 Descripción de la memoria 2

2. ESTADO DE LA MATERIA DE LA ROBÓTICA 3

2.1 Clasificación de robots 3

2.2 Robot manipulador o industrial. 4

2.2 Robot colaborativo. 6

3. SISTEMA OPERATIVO ROBÓTICO 8

3.1 Introducción a ROS 8

3.2 ¿Por qué el uso de ROS? 10

3.3 Arquitectura y funcionamiento 11

3.3 Herramientas… 13

3.4.1 Gazebo 13

3.4.2 RViz ….. 14

3.4.3 URDF 15

3.4.4 MoveIt 16

4. GUÍA DE INICIACIÓN EN ROS 17

4.1 Instalación de ROS Kinetic 17

4.2 Configuración de ROS 19

4.3 Primeras pruebas de funcionamiento 21

5. CONTROL DE ROBOTS INDUSTRIALES ABB 28

5.1 Robots ABB 28

5.2 Configuración del robot IRB 120 con MoveIt 30

5.3 Planificación de movimientos 36

5.3.1 Movimiento por comandos 36

5.3.2 Programación de movimiento en Python 38

5.3.3 Movimiento con MoveIt y RViz 39

5.3 Simulación multirobot 42

6. TAREAS DE MANIPULACIÓN CON ROBOT COLABORATIVO 45

6.1 Universal Robots. 45

6.2 Manipulación con pinza robótica. 47

6.3 Manipulación con "vacuum gripper". 51

7. CONCLUSIONES Y LINEAS FUTURAS 57

7.1 Conclusiones. 57

7.2 Líneas futuras. 58

BIBLIOGRAFÍA. 60

ANEXOS 61

A Características técnicas del robot industrial IRB120. 61

B Características técnicas del robot colaborativo UR5 63

C Solución de errores 64

D Código para tarea de manipulación con pinza. 71

E Código para tarea de manipulación con "vacuum gripper" 75

LISTA DE FIGURAS. 80

1. Introducción

1.1 MOTIVACIÓN Y OBJETIVOS

Este trabajo comenzó sin que yo me diese cuenta varios años atrás, cuando estuve trabajando de becario en el departamento de métodos en Valeo Térmico Zaragoza. Observé la cantidad excesiva de tiempo que se perdía cuando se estropeaba o se tenía que programar un nuevo robot en una línea de montaje. Cuando sucedía un imprevisto, tanto los metodistas, como los encargados de mantenimiento, tenían que acordarse o ir aprendiendo sobre la marcha como se programaba cada robot, ya que al ser fabricados por diferentes empresas su programación no era la misma.

Es por ello, por lo que emprendí la búsqueda de información, acerca de si estaba implementado algún tipo de sistema operativo, que pudiese controlar cualquier tipo de robot sin importar la marca ni el modelo. Descubrí que sí, que estaba implementado y su nombre era “Robot Operating System”

Por este motivo, cuando encontré en la infinita lista donde se ofertan los Trabajos de Fin de Grado, un trabajo en el cual tenía la oportunidad de aprender ROS, no dudé en adjudicármelo. Tras unas pequeñas modificaciones de la propuesta inicial, lo adapté a lo que yo había estado trabajando, que eran los robots manipuladores.

El objetivo principal de este proyecto es el partir de una base cero, y llegar a ser capaz de realizar tareas de manipulación de un robot mediante el Sistema Operativo Robótico y las diferentes herramientas que ofrece. Para ello se irá paso a paso, aprendiendo desde lo más básico, como puede ser desde la instalación del programa, hasta intentar alcanzar el objetivo principal. Por supuesto, dicho objetivo no es el único, ya que el conseguir esta meta tiene por el camino otros objetivos secundarios que se describen a continuación:

- Familiarización, aprendizaje y uso avanzado de Linux.
- Aprendizaje del funcionamiento y desarrollo de sistemas basados en ROS.
- Adquisición de los conocimientos básicos del lenguaje de programación Python.
- Familiarización, aprendizaje y uso avanzado de las herramientas necesarias para simular un robot con ROS (Gazebo, RViz y MoveIt).

1.2 DESCRIPCIÓN DE LA MEMORIA

Tal y como se acaba de mencionar, el principal objetivo de este Trabajo Fin de Grado es el de ser capaz de programar una tarea de manipulación en un entorno de simulación. Para ello se ha redactado una memoria dividida en siete capítulos.

Tras la presente introducción, en el Capítulo 2 se enfoca el proyecto mediante un estudio de la materia acerca de los distintos tipos de robots. Se hace especial hincapié en los tipos de robot que en capítulos posteriores serán utilizados.

En el Capítulo 3 se describe todo el entorno del Sistema Operativo Robótico, desde su definición más básica, hasta las herramientas con las que trabaja. Este capítulo sirve de introducción para lo que se verá en el siguiente.

Seguidamente en el Capítulo 4, se comienza con la práctica y con el uso de ROS. Para ello se describe su instalación, su configuración y para comprobar que ambas han sido satisfactorias se realizan 3 pruebas básicas de funcionamiento.

La primera simulación de un robot manipulador se realiza en el Capítulo 5. Se realiza una breve descripción de la marca (ABB) y el modelo (IRB 120) del robot utilizado. Este modelo se configura con la herramienta "MoveIt" y se ejecutan varios movimientos con distintas plataformas. Por último, se simulan dos robots en un mismo entorno.

En el Capítulo 6 se llevan a cabo dos tareas de manipulación mediante un robot colaborativo. Para la comprobación de que un mismo programa puede ser utilizado en ROS para trabajar con diferentes robots, parte del código del Capítulo anterior es utilizado, pero con el modelo UR5 de la empresa Universal Robots. Primeramente, son descritas las características principales de estos robots colaborativos. A continuación, se describe ambas tareas, cuya gran diferencia es la herramienta utilizada para el transporte de las piezas (una pinza robótica y un "vacuum gripper").

El Capítulo 7 contiene las conclusiones desprendidas del trabajo y las posibles líneas futuras de investigación.

2. Estado de la materia de la robótica

En este Capítulo, se introduce al mundo de la robótica mediante un estado de la materia que describe los distintos tipos de robot. Se hace hincapié en los tipos de robot con los cuales se trabaja en apartados posteriores.

2.1 CLASIFICACIÓN DE LOS ROBOTS

Existen numerosas clasificaciones para distinguir un robot de otro, pero en este caso se destacan dos de las más importantes. La primera es dividirlos según su generación, es decir por su cronología.

-1º Generación: Son los llamados robots manipuladores cuyo sistema de control se basa en paradas fijas. Utilizados para la repetición de tareas programadas.

-2º Generación: Los robots de segunda generación o robots de aprendizaje obtienen una información limitada del entorno y se controlan mediante una secuencia numérica. Son de mayor tamaño que sus predecesores y se utilizan principalmente en la industria automovilística.

-3º Generación: Es el comienzo de los robots inteligentes debido a que son capaces de adquirir información de su entorno mediante diversos sensores. Se controlan mediante ordenadores y se comienza a desarrollar lenguajes de programación**.**

-4º Generación: Los robots ya son capaces de controlar su entorno automáticamente gracias a que se ajustan y se mejoran sus sistemas sensoriales.

-5º Generación: Época actual, en la cual los robots están ligados a la inteligencia artificial. Gracias a los avances tecnológicos, los robots son cada vez más autónomos y se intenta que tengan capacidades de cubrir las necesidades humanas.

La otra gran distinción es dividir a los robots dependiendo de la función que realicen:

-Militares: Su función es la de asistir a los ejércitos en diferentes operaciones como la de desactivación de minas, rastrear, búsqueda de personas etc.

-De servicios: Dispositivos autónomos controlados por ordenador. Estos tipos de robots se dividen en diferentes subgrupos, por ejemplo, los robots domésticos (el más conocido es el aspirador “Roomba”) o los robots de investigación (utilizados en laboratorios o en el ámbito aeroespacial).

-Médicos: En esta categoría se incluyen todos los robots diseñados para el ámbito de la medicina. Desde robots utilizados en cirugía, como diseñados para asistir a personas con diversidad funcional, incluso prótesis de última generación.

-Industriales: Robots cuya función es la de realizar manipulaciones automáticas durante las distintas fases de la producción industrial. Son robots reprogramables y multifuncionales con los que habitualmente se busca reducir el tiempo de fabricación [1] .

Existen otras clasificaciones como puede ser: según su nivel de inteligencia (manejo manual, regeneradores, inteligentes etc.), según su entorno de trabajo (fijos, aéreos, submarinos etc.) o según su estructura, tal y como queda reflejado en la Figura 2.1.

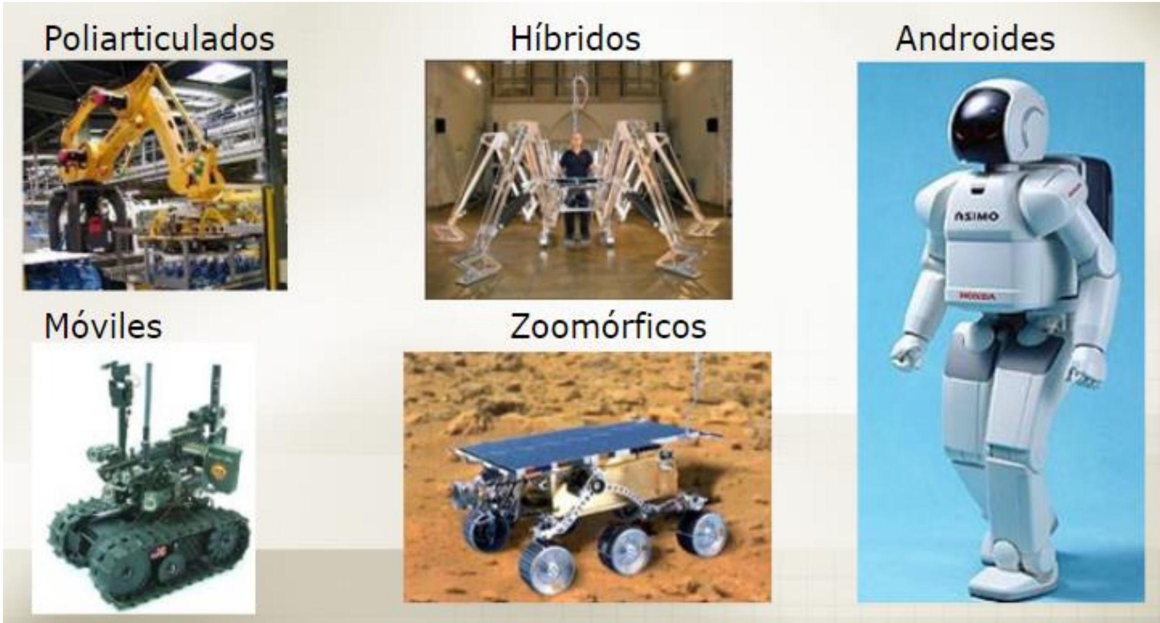


Figura 2.1: Clasificación de los robots dependiendo de su estructura [2].

2.2 ROBOT MANIPULADOR O INDUSTRIAL

Según define la norma UNE-EN ISO 10218-1, un robot industrial es un manipulador controlado automáticamente, reprogramable y multifuncional, programable en tres o más ejes, que puede ser fijo o móvil y que se utiliza en aplicaciones industriales automatizadas.

Los diferentes tipos de robots industriales son los siguientes: Robot cartesiano, Scara, cíclico, esférico, angulares (o de seis ejes) y paralelos. Esta diferenciación se debe a la manera en que ejecutan sus movimientos, como se puede visualizar en la Figura 2.2.

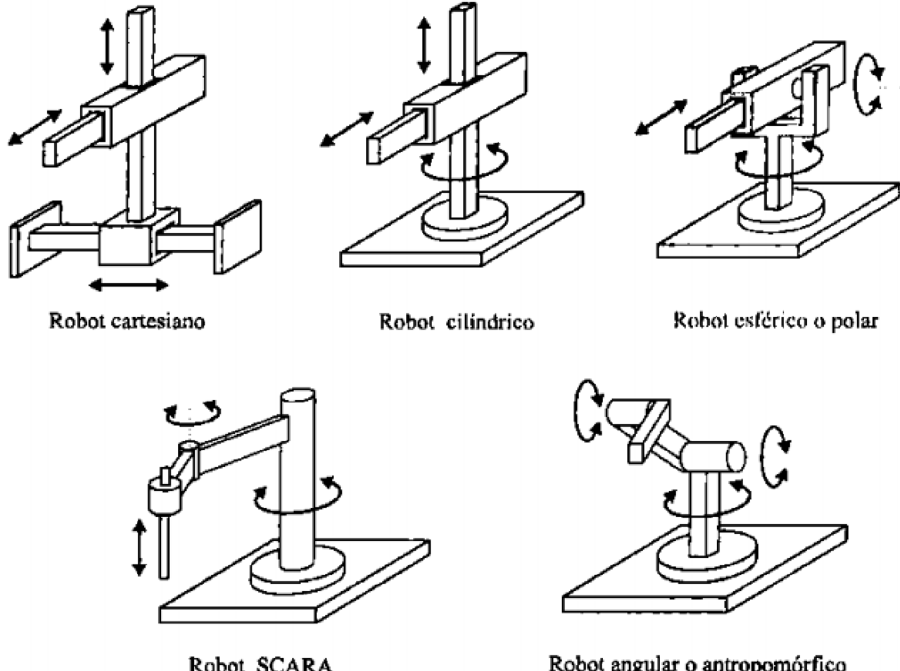


Figura 2.2: Clasificación de los robots industriales en función de sus movimientos [3].

Generalmente no trabajan de forma independiente sino en conjunto con otras máquinas y herramientas formando células de trabajo y su utilización en la industria se debe a los siguientes factores: Fácil reprogramación, ahorro de costes, reducción del tiempo de producción, mayor seguridad del personal, producción más flexible, mejora del flujo de datos etc.

Por todos estos factores los robots cuentan con innumerables aplicaciones como pueden ser:

- Operaciones de procesamiento, como soldadura, pintura, etc.
- Operaciones de ensamblaje, donde el trabajo repetitivo facilita el uso de este tipo de robot.
- Operaciones de empaque (en tarimas o pallets), agilizando el proceso y manejando grandes pesos.
- Operaciones de soldadura (por arco, por puntos, por gas etc.)
- Otro tipo de operaciones como pueden ser remachados, estampados, cortes mecánicos, sistemas de medición, controles de calidad etc [4].

De todos los tipos nombrados anteriormente los más utilizados son los robots de seis ejes o también llamados robots angulares. Por lo tanto, serán estos robots los que se simularán en los siguientes capítulos para su manipulación.

La característica principal es que la constitución física de la mayor parte de estos robots industriales, guarda cierta similitud con los brazos del cuerpo humano. Por este motivo, para hacer referencia a los distintos elementos que componen el robot, se usan términos como cintura, hombro, brazo, codo, muñeca, etc. En la Figura 2.3 se puede apreciar dicho parecido. Esta similitud les permite realizar infinidad de movimientos y poder llevarlos a cabo con rapidez y gran precisión.

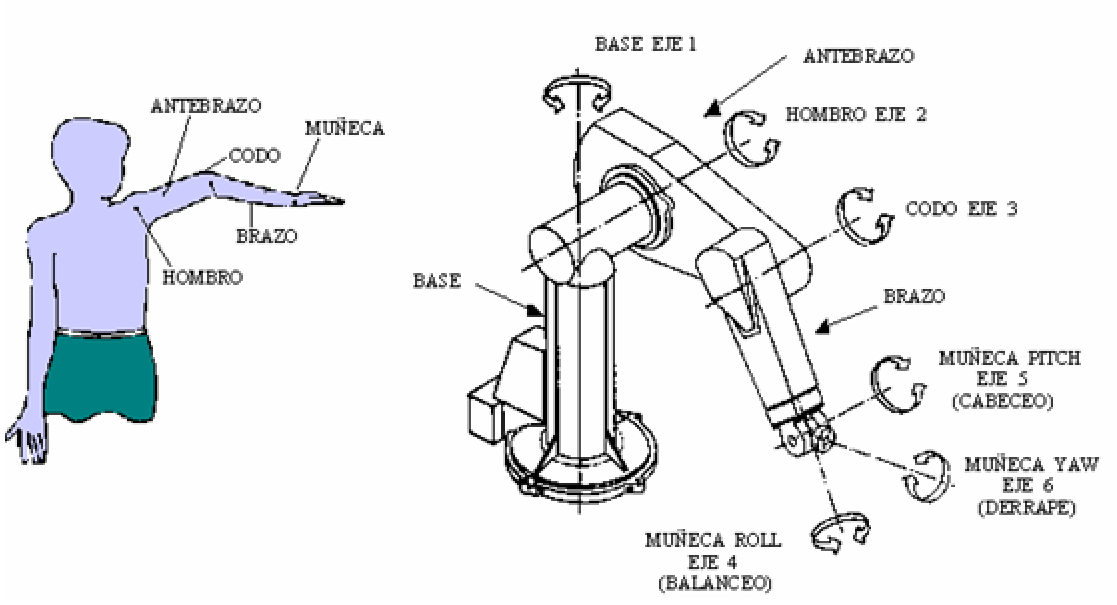


Figura 2.3: Semejanza entre un brazo humano y un robot industrial de seis ejes [5].

2.3 ROBOT COLABORATIVO

La utilización de robots en los procesos de producción tiene una larga trayectoria en el mundo industrial, pero en la actualidad hay que hablar de un nuevo concepto de robótica: la robótica colaborativa. Así, nos encontramos una nueva forma de trabajar con robots, en la que personas y robots trabajan mano a mano en un mismo espacio, sin que existan barreras físicas entre ellos.

Un robot colaborativo o cobot se define en la norma UNE-EN ISO 10218-2 como un robot diseñado para interaccionar directamente con una persona dentro de un espacio de trabajo colaborativo. Dicho trabajo es el espacio de trabajo en el que, durante su funcionamiento, el robot y la persona pueden desarrollar tareas de forma simultánea [6].

Es importante mencionar la seguridad que los robots colaborativos aportan a los

procesos industriales. A diferencia de los robots industriales tradicionales, no necesitan vallado de seguridad, ya que están preparados para trabajar junto a los operarios en los entornos de producción.

Los cobots están programados para detenerse en el momento en el que entran en contacto. Basta que dicho contacto sea mínimo para que el cobot se detenga y deje de realizar la función para la que ha sido programado.

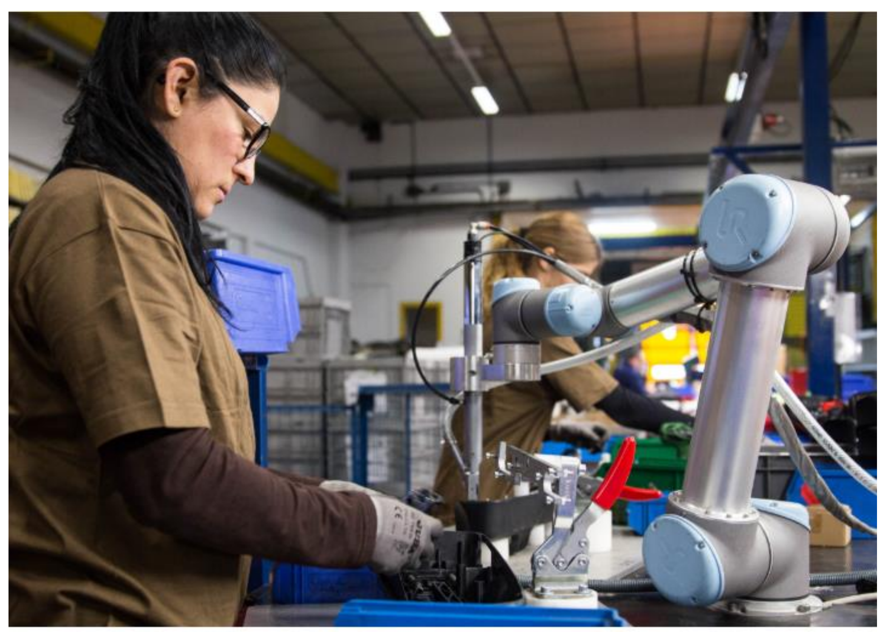


Figura 2.4: Operaria y cobot comparten espacio de trabajo en la industria [6].

La empresa puntera en la fabricación y puesta a punto de estos robots colaborativos es “Universal Robot”. Esta empresa danesa fabrica tres modelos diferentes que posteriormente en el Capítulo 6 se describen con más detalle.

3. Sistema operativo robótico

En este tercer Capítulo se presenta el sistema operativo robótico (ROS). Se describe tanto su funcionamiento, como los motivos de su utilización. Por último, se hace una breve descripción de las herramientas más importantes con las que trabaja ROS.

3.1 INTRODUCCIÓN A ROS

ROS son las siglas en inglés de "Robot Operating System". Se trata de un "framework" (o infraestructura digital) flexible de código abierto, que consta de una colección de herramientas, bibliotecas y convenios, y que tienen como objetivo simplificar la tarea de desarrollar un software para robots. Un "framework" es un conjunto de librerías, utilidades y métodos de programación que permiten el desarrollo de aplicaciones orientadas a cierta área concreta como puede ser: diseño web, visión artificial, robótica, etc.

A pesar de su nombre, ROS no es un sistema operativo en sí, como demuestra el hecho de que requiere ser instalado sobre otro (preferentemente sobre sistemas Linux, como Ubuntu). Sí que proporciona servicios estándares como abstracción del hardware, comunicación a través de mensajes y gestión de archivos, por nombrar algunos, por lo tanto, se le conoce como un Meta-Sistema Operativo.

La filosofía de ROS puede resumirse en las siguientes características:

- **Multilenguaje:** Admite varios lenguajes de programación populares en el campo de la robótica. En la actualidad es capaz de soportar tres muy distintos, como son C++, Python y Lisp, y cuenta con librerías experimentales en Java, Ruby y Lua.
- **Basado en herramientas***:* Cuenta con un diseño donde un gran número de herramientas primitivas son llamadas para crear y ejecutar los diversos componentes de ROS, dejando la gestión de memoria, sistema de archivos, etc. al sistema operativo.
- **Ligero:** Todo el código está estructurado en pequeñas bibliotecas independientes, de tal manera que es mucho más fácil reutilizarlo en proyectos distintos al original para el que fueron creados.

- **Proceso distribuido**: Se programa en forma de unidades mínimas de procesos (nodos), y cada proceso se ejecuta de forma independiente e intercambia datos sistemáticamente.
- **Gratuito y de código abierto:** ROS es un software libre bajo licencia estándar BSD (Berkeley Software Distribution) de tres cláusulas donde todos los paquetes son públicos.
- **Gran cantidad de componentes:** ROS está rodeado de todo tipo de componentes, que le permiten ampliar sus aplicaciones (comunicación, simulación etc.) Estos componentes se desglosan en la Figura 3.1.

Es importante destacar este último apartado ya que ROS, como todos los recursos que este ofrece, son de código abierto (OSS), siendo su uso totalmente gratuito. Esta fue, de hecho, la filosofía más importante que impulsó su creación de su primera versión en 2010 (ROS Box Turtle): una plataforma que permitiera el desarrollo de sistemas robóticos de forma mucho más óptima gracias a la aportación de todos sus usuarios. Es posible compartir todo tipo de herramientas y programas, de forma libre, reutilizables, adaptables a las necesidades de cada uno e integrables con todo el software robótico existente y futuro [7].

Figura 3.1: Componentes de ROS [8].

3.2 ¿POR QUÉ EL USO DE ROS?

En primer lugar, tal y como se alude en la introducción, la principal motivación de este proyecto era investigar sobre un Meta-Sistema Operativo, capaz de simular todo tipo de robots sin necesidad de saber el lenguaje de programación de cada uno de ellos. Se verá más adelante que con un mismo algoritmo se pueden programar movimientos de robots fabricados por distintas empresas. Otra razón de peso es que tanto ROS, como todo el material asociado, son de código libre, por lo que los gastos para su utilización son nulos.

Sin ser menos importante, cabe mencionar que ROS y sus aplicaciones proporcionan un entorno ideal para el desarrollo de todo tipo de aplicaciones robóticas reales. Es decir, tiene una clara orientación al mundo real, lejos de simulaciones en las que sólo se busque algún resultado de carácter gráfico. Si bien lo desarrollado para este proyecto, ha sido una serie de algoritmos que permiten la simulación de robots manipuladores y colaborativos, gracias al haberlo desarrollado en ROS sólo se requieren unos mínimos cambios en dichos algoritmos para posibilitar la experimentación real.

Otro de los aspectos a tener en cuenta de ROS es que admite diferentes lenguajes de programación, como C++ o Python, siendo este último el que se ha usado en el presente proyecto. En la Figura 3.2 se observan los distintos lenguajes que admite ROS, así como todo el ecosistema que lo envuelve.

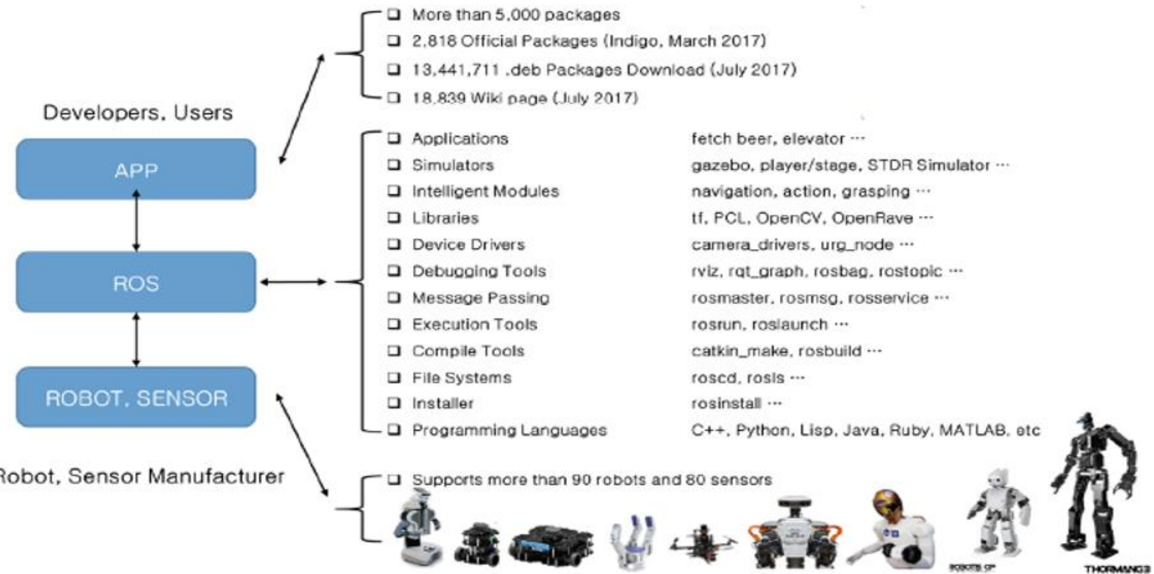


Figura 3.2: Ecosistema que rodea a ROS [8].

3.3 ARQUITECTURA Y FUNCIONAMIENTO

ROS presenta una estructura modular, en la que los programas se crean a partir de paquetes, los cuales contienen los archivos necesarios para desarrollar las aplicaciones que requiere el usuario. Esta estructura a la vez que su funcionamiento se basa principalmente en los siguientes elementos:

- **Paquetes**: Los paquetes son elementos que permiten agrupar una serie de librerías o aplicaciones, facilitando su portabilidad e instalación en diferentes equipos. Su estructura se expone en la Figura 3.3.

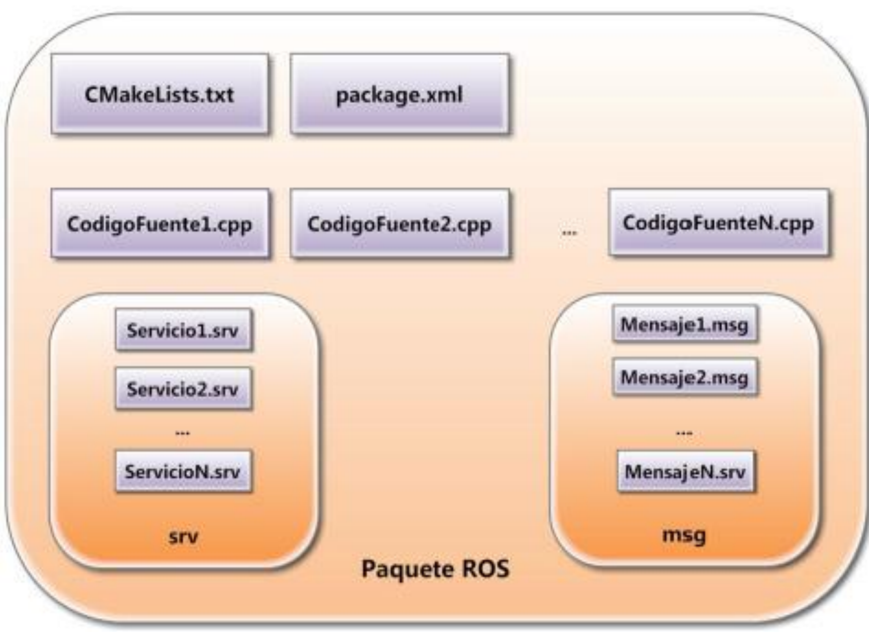


Figura 3.3: Estructura básica de un paquete ROS [9].

- **Nodos:** La unidad básica computacional de este software es el nodo, en él se produce toda la computación además de encargarse de gestionar la información tanto entrante como saliente. Cada uno de ellos tiene asociado un script (escrito en Python o C++) que le dicta su configuración y su comportamiento. Siempre ha de haber un nodo maestro que se encarga de llevar un registro de los nodos dados de alta y de baja. Un ejemplo de su funcionamiento se muestra en la Figura 3.4.
- **Nodo maestro:** Es el nodo que se encarga de proporcionar el registro de nombres, permitiendo así al resto de nodos del sistema comunicarse entre sí, intercambiando mensajes, solicitando servicios, etc.

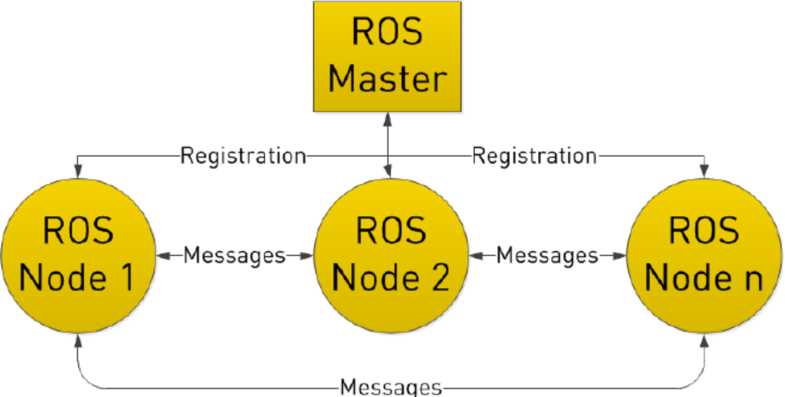


Figura 3.4: Sencillo esquema del sistema de funcionamiento entre nodos [9].

- **Mensajes:** Estructura de datos encargada de permitir la comunicación entre los distintos nodos. Esta información es enviada a través de los *"topics"*.
  ROS tiene definido muchos tipos de mensajes, algunos ejemplos de mensajes son int8 (entero de 8 bits) o float64 (flotante de 64 bits).
- **"Topics":** Es el nombre que se usa para identificar y poder encaminar los distintos mensajes dentro de la red ROS. Así, un nodo envía mensajes publicándolos en un determinado "topic" y, de la misma forma, un nodo recibe mensajes suscribiéndose al "topic" correspondiente. En función de si un nodo publica o recibe mensajes de un tópico, se considera "*Publisher"* o "*Subscriber"*.
- **Servicios:** Cuando se publica en un "topic", se están enviando datos entre diferentes nodos sin ningún tipo de control. Sin embargo, cuando se necesita una solicitud o respuesta de un nodo es necesario emplear servicios. Los servicios, por tanto, proporcionan el modelo de petición y respuesta que a menudo se necesita en este tipo de sistemas distribuidos.
- **Lanzadores:** Es posible ejecutar nodos, llamar a servicios y leer o publicar en "topics" directamente a través de la línea de comandos de la terminal. Sin embargo, si lo que deseamos es lanzar varios nodos a la vez, cargar una serie de parámetros o ejecutar varias herramientas, existe una forma más sencilla de hacerlo, sin tener que hacerlo uno a uno. Para ello ROS contiene una herramienta denominada "*roslaunch"* que permite, por ejemplo, lanzar varios nodos con sus argumentos, tomándolos de un archivo con extensión .launch [10].

3.4 HERRAMIENTAS

3.4.1 Gazebo

Gran parte de la utilización de ROS se debe a que ofrece una potente herramienta de simulación. Esta herramienta llamada Gazebo es un simulador 3D, que nos permite probar rápidamente algoritmos, robots diseñados y realizar pruebas de regresión utilizando escenarios realistas. Al tener la capacidad de controlar los entornos sobre los que los robots van a funcionar, permiten probar el funcionamiento de éstos en un escenario controlado antes de llevarlo a la realidad.

Gazebo es uno de los simuladores de código abierto más populares en los últimos años, debido a su alto rendimiento y a su gran compatibilidad con ROS. Esto se debe a que es desarrollado y distribuido por Open Robotics, que está a cargo de ROS y su comunidad.

A continuación, se describen algunas de las características más importante de este simulador 3D [11]:

- Utiliza varios motores físicos como Bullet, Simbody y DART para lograr una simulación más dinámica.
- Gazebo utiliza OGRE (Motores de renderizado de gráficos de código abierto) para sus gráficos 3D.
- Son compatibles numerosos sensores como cámaras 2D / 3D, cámaras de profundidad, sensor de contacto, simulación de ruido, etc.

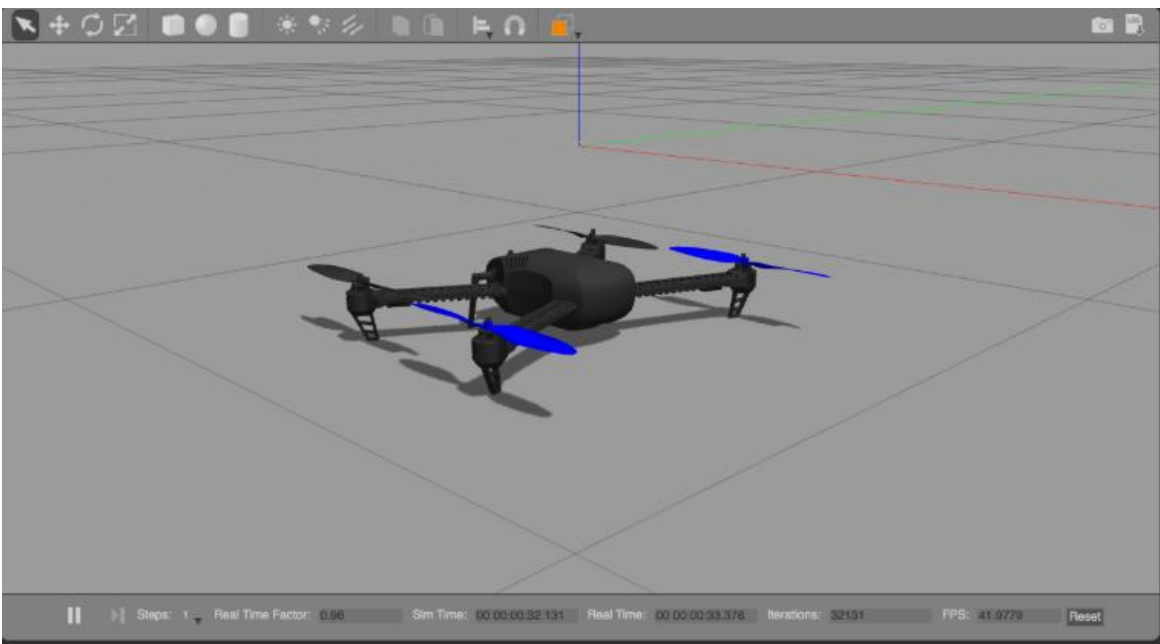


Figura 3.5: Simulación de un dron en Gazebo [11].

- Proporcionan varias API (Interfaz de programación de aplicaciones) para permitir a los usuarios crear robots, sensores y control del entorno o bien usar modelos de robot ya creados como TurtleBot, iRobot o PR2. Un ejemplo de modelo se muestra en la Figura 3.5.
- Se admiten herramientas desde la línea de comandos para verificar y controlar el estado de la simulación.

3.4.2 RVIZ

RViz es la abreviación de "ROS Visualization" y es una de las herramientas más útiles y completas que ofrece ROS, ya que permite mostrar en 3D de manera gráfica, intuitiva y en tiempo real, el estado y la información de un sistema en ROS. En ella se pueden integrar todo tipo de elementos (visión, control, navegación…) y controlarlos de forma interactiva.

Se puede generar un modelo y escribirlo en formato URDF (Unified Robot Description Format) donde se especifican las dimensiones del robot, parámetros físicos como masa e inercia, los movimientos de las articulaciones, "joints", "links" etc. Utilizando la herramienta RViz se puede simular dicho modelo y añadirle sensores o ejecutar movimientos. Aparte de visualizar todo tipo de robots, también se puede conectar a un robot real y reproducirlo en el ordenador respondiendo en tiempo real a sus movimientos y sensores.

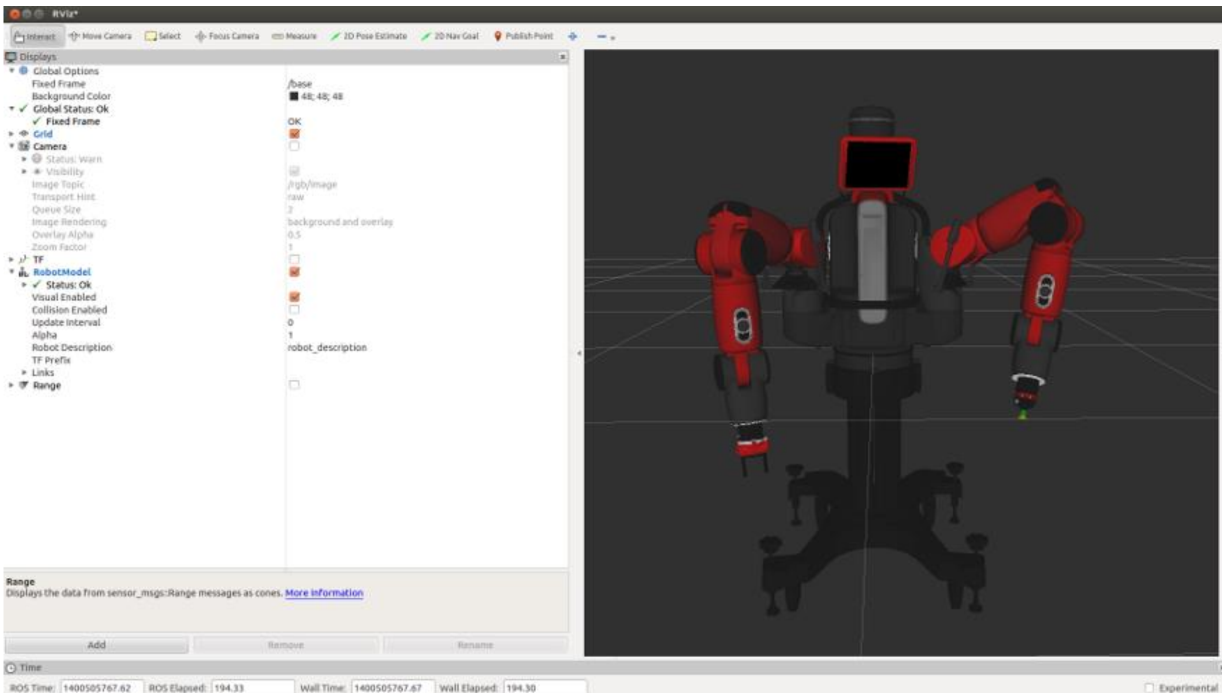


Figura 3.6: Visualización de un robot en RViz [12].

Por nombrar alguna de sus múltiples funciones, se encuentra la posibilidad de ver la lectura de los sensores como cámaras de visión o láseres, visualizar la trayectoria que va a seguir dentro de un mapa, mostrar la configuración de las distintas coordenadas del robot, etc. Estas funciones se encuentran distribuidas en su interfaz, la cual se visualiza en la Figura 3.6.

3.4.3 URDF

URDF (Unified Robot Description Format) es una herramienta de modelado de robots. Se encarga de especificar las propiedades del robot, como pueden ser sus dimensiones, número de articulaciones, parámetros físicos, etc.

Toda la información viene descrita en forma de árbol en un archivo XML, distinguiéndose dos componentes principales necesarios para la construcción de la cadena cinemática de un robot, los eslabones (“Links”) y las articulaciones (“Joints”).

•“Links”: Constituyen las componentes físicas del robot, es decir, cada uno de los eslabones o enlaces por lo que está formada la parte rígida del robot, como la masa, inercia o geometría. También compone la parte visual para mostrar el robot en herramientas como RViz

•“Joints”: Son las articulaciones o uniones mediante los cuales se establece la relación entre los distintos eslabones o “links” del robot. Describen además la cinemática y dinámica de cada articulación, además de especificar los límites de colisión del robot.

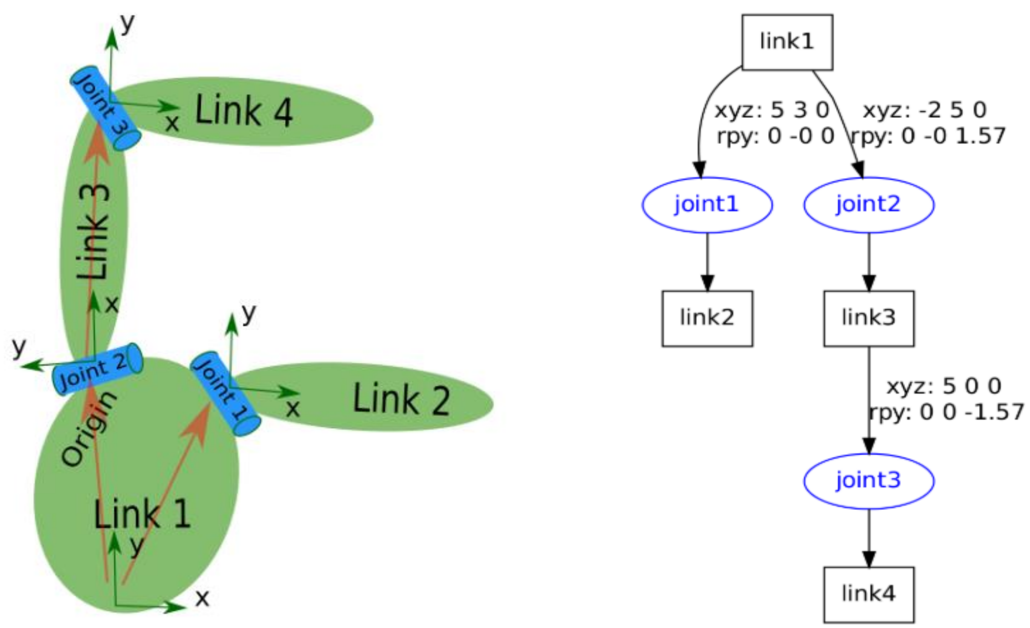


Figura 3.7: Dos esquemas de los distintos componentes de un fichero URDF [13].

Dos ejemplos de visualización de los componentes de un robot en forma de árbol con sus diferentes "joints" y "links" se observan en la Figura 3.7.

3.4.4 MoveIt

MoveIt! es una librería de código abierto para manipuladores que proporciona una variedad de funciones. Algunas de las más destacadas son el análisis rápido de cinemática inversa para planificación de movimiento, algoritmos avanzados para manipulación, control manual de robot, dinámica y controladores.

Para ser capaces de trabajar con cualquier robot, MoveIt incorpora un asistente que nos permite configurar todos los detalles necesarios para generar los archivos, que le servirán al paquete para reconocer y comunicarse con dicho robot. El asistente trabaja a través de una interfaz gráfica que hace más llevadero este proceso. En el Capítulo 5.2 se redacta un ejemplo de cómo utilizar este asistente, describiendo todas sus utilidades.

En la Figura 3.8, se detalla la estructura del nodo principal que ofrece Moveit, llamado "move_group". Este nodo sirve de integrador, enlazando todos los componentes individuales en un cuerpo que puede ofrecer un conjunto de acciones ROS y servicios para el usuario [14].

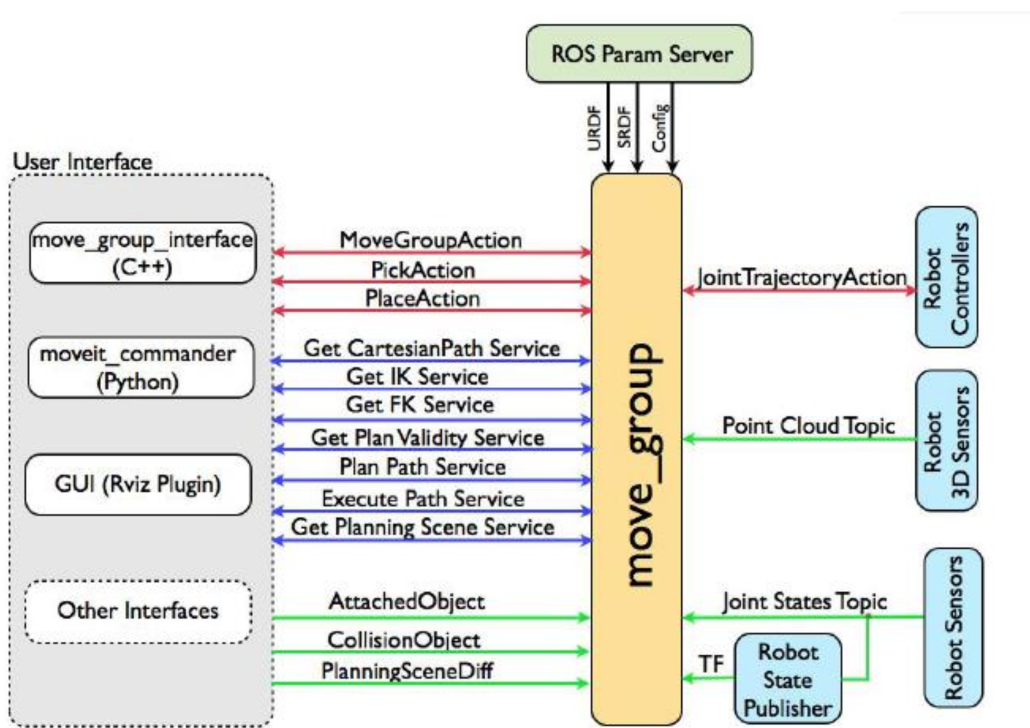


Figura 3.8: Estructura de comunicación del nodo "move_group" [8].

4. Guía de iniciación en ROS

En este Capítulo cuatro se comienza a utilizar ROS. Para ello, se procede a su instalación y a su configuración. Para comprobar que tanto el funcionamiento de ROS, como el de sus herramientas es correcto, se realizan tres pruebas diferentes.

4.1 INSTALACIÓN DE ROS KINETIC.

ROS es un Meta-Sistema Operativo creado para ser instalado en el sistema operativo "Linux", en sus diferentes versiones. También existe la posibilidad de que se pueda utilizar tanto en Windows como en MAC a través de máquinas virtuales como "Virtual Box" o "VMware".

ROS cuenta con innumerables versiones, ya que desde su lanzamiento en 2010 es implementada una nueva versión por año. Las diferencias entre ellas acostumbran a ser temas de arquitectura, mejoras y corrección de errores. La gran diferencia es que cada versión se lanza para unas distribuciones específicas del sistema operativo (por ejemplo, Kinetic para Ubuntu 15.10 y 16.04, Lunar para Ubuntu 16.10 y 17.04).

Se ha elegido trabajar con el sistema operativo Ubuntu 16.04 Xenial Xerus y con ROS Kinetic Kame, por dos motivos principales. El primero, es que esta versión fue lanzada en 2016, por lo tanto, está comprobado que su funcionamiento es estable y correcto en comparación con las últimas versiones de ROS. El segundo motivo se debe a que los robots simulados en capítulos posteriores (ABB y Universal Robots), tienen paquetes que se han comprobado que funcionan con la versión Kinetic.

Tras la recopilación de información entre varios tutoriales, a continuación, se explican todos los **pasos seguidos para la instalación de ROS de la manera más sencilla y rápida posible.**

I. En primer lugar, se abre una ventana de la terminal ejecutando "CTRL+ALT+T" y se añade al gestor de paquetes de Ubuntu la dirección para obtener los paquetes ROS y sus actualizaciones, para ello es necesario editar el archivo /etc/apt/sources.list.d:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

II. Se agrega una clave pública para poder descargar el paquete del repositorio ROS con el siguiente comando:

$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116

III. Una vez se ha puesto la dirección del repositorio ROS en la fuente, se recomienda actualizar todos los paquetes de Ubuntu instalados antes de instalar ROS. Tras realizar estos tres primeros pasos la terminal muestra el aspecto de la Figura 4.1.

$ sudo apt-get update

```
inazio@inazio-Aspire-V3-371: ~
inazio@inazio-Aspire-V3-371:~$ sudo sh -c 'echo "deb http://packages.ros.org/ros
/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
[sudo] password for inazio:
inazio@inazio-Aspire-V3-371:~$ sudo apt-key adv --keyserver 'hkp://keyserver.ubu
ntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
Executing: /tmp/tmp.PmpHpoxXbG/gpg.1.sh --keyserver
hkp://keyserver.ubuntu.com:80
--recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
gpg: solicitando clave AB17C654 de hkp servidor keyserver.ubuntu.com
gpg: clave AB17C654: clave pública "Open Robotics <info@osrfoundation.org>" impo
rtada
gpg: Cantidad total procesada: 1
gpg:               importadas: 1  (RSA: 1)
inazio@inazio-Aspire-V3-371:~$ sudo apt-get update
Obj:1 http://es.archive.ubuntu.com/ubuntu xenial InRelease
Des:2 http://packages.ros.org/ros/ubuntu xenial InRelease [4.678 B]
Des:3 http://packages.ros.org/ros/ubuntu xenial/main amd64 Packages [795 kB]
Des:4 http://packages.ros.org/ros/ubuntu xenial/main i386 Packages [567 kB]
Descargados 1.367 kB en 8s (158 kB/s)
Leyendo lista de paquetes... Hecho
inazio@inazio-Aspire-V3-371:~$
```

Figura 4.1: Captura de pantalla de la terminal después de haber realizado los tres primeros pasos.

IV. Llegados a este punto se presentan diferentes opciones de cara a la instalación. Se puede instalar el sistema completo con todos los paquetes, una versión de escritorio que ocupa menos espacio, pero no contiene la versión entera *($ sudo apt-get install ros-kinetic-rqt**), o también se puede instalar uno a uno los paquetes individuales que se necesiten ($ sudo apt-get install ros-kinetic-[NAME_OF_PACKAGE]). En este caso se opta por

la instalación de la versión completa, ya que además de contener ROS también cuenta con rqt, RViz, bibliotecas relacionadas con robots, simulación, navegación, etc.

```
$ sudo apt-get install ros-kinetic-desktop-full
```

V. Antes de poder utilizar ROS es necesario instalar e inicializar “rosdep”. Es una dependencia del sistema que mejora la instalación sencilla de paquetes dependientes y utilizar de forma conjunta todos los paquetes que tengamos instalados en el sistema. Se ejecutan los siguientes comandos:

```
$ sudo rosdep init
$ rosdep update
```

VI. Por último, se instalará el programa “rosinstall”. Es una herramienta que permite con un comando descargar gran cantidad de paquetes de software e instalarlos.

```
$ sudo apt-get install python-rosinstall
```

4.2 CONFIGURACIÓN DE ROS

Antes de comenzar cualquier prueba con ROS se necesita configurar el entorno, lo cual se describe a continuación en dos pasos bien diferenciados.

Configuración del entorno de desarrollo: Para la configuración es necesario que se explique que es el “bashrc”. Se trata de un archivo que carga todas las rutas y comandos que Linux puede reconocer en base a todos los paquetes o programas instalados. Es decir, cada vez que se abra un terminal, todo lo que venga especificado en el archivo “bashrc” será ejecutado. Es importante tener claro el contenido de este, ya que, si al instalar un nuevo paquete no tenemos la precaución de definir en el “bashrc” que ejecute el comando “source”, puede crear problemas durante la ejecución del programa, al no encontrarlo en sus listas de comandos disponibles. Con el siguiente comando se importa el archivo de configuración del entorno.

$ source /opt/ros/kinetic/setup.bash

Este comando se tendría que escribir cada vez que se abre una terminal, pero para evitar esta tarea repetitiva, se puede configurar para importar un archivo de configuración cada vez que abramos una nueva ventana de terminal. Además, se puede configurar la red ROS y crear comandos rápidos para los comandos de uso frecuente. Se utiliza el programa "gedit", que es un editor de texto para abrir el archivo ".bashrc". Una vez abierto el archivo se debe escribir lo mencionado anteriormente, por lo tanto, la secuencia de comandos sería la siguiente:

$ gedit ~ / .bashrc (se abre el archivo con el editor de texto)
$ source /opt/ros/kinetic/setup.bash (línea de código que se tiene que escribir)
$ source ~/.bashrc (se guarda la configuración)

Creación de un entorno de trabajo: Para poder compilar los paquetes de ROS, lo primero es configurar un espacio de trabajo o "workspace", donde residen todos los archivos con los que se trabaje en un futuro. Un espacio de trabajo es, simplemente, un conjunto de directorios o carpetas en las cuales se encuentran agrupados todos los códigos ROS. Pueden existir múltiples "workspace", pero sólo se puede trabajar en uno de ellos simultáneamente, como se verá más adelante en el Capítulo 6.3.

Se utiliza un sistema de construcción de ROS llamado "catkin". Se abre una terminal y se escriben las siguientes líneas de comandos:

$ mkdir -p ~/catkin_ws/src (crea el directorio llamado "catkin_ws" y dentro una carpeta de nombre "src")
$ cd ~/catkin_ws/src (accede a la carpeta recién creada)
$ catkin_init_workspace (inicia la carpeta)

Actualmente, el espacio de trabajo de "catkin" solo contiene la carpeta "src" y el archivo "CMakeLists.txt" dentro. Se ejecuta:

$ cd ~/catkin_ws/ && catkin_make (accede a la carpeta y compila el código del archivo "CMakeLists.txt")

Cuando termina de compilar sin errores, se puede observar que además de la carpeta "src" constituida anteriormente, se han creado las carpetas "build" y "devel". Los archivos relacionados con la compilación, para el sistema "catkin" se guardan en la

carpeta "devel", mientras que los archivos relacionados con la ejecución se guardan en la carpeta "build". Para configurar estas carpetas, se repiten las acciones de abrir con el editor de textos el archivo ".bashrc", e introducir la siguiente línea de código:

$ source ~/catkin_ws/devel/setup.bash

```
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
  if [ -f /usr/share/bash-completion/bash_completion ]; then
    . /usr/share/bash-completion/bash_completion
  elif [ -f /etc/bash_completion ]; then
    . /etc/bash_completion
  fi
fi


source /opt/ros/kinetic/setup.bash
source ~/catkin_ws/devel/setup.bash
# Set ROS Network
export ROS_HOSTNAME=localhost
export ROS_MASTER_URI=http://localhost:11311
# Set ROS alias command
alias cw='cd ~/catkin_ws'
alias cs='cd ~/catkin_ws/src'
alias cm='cd ~/catkin_ws && catkin_make'
```

Figura 4.2: Últimas líneas del código del archivo ".bashrc".

Como se puede observar en la Figura 4.2, las últimas líneas del archivo sirven para configurar la red ROS. En este caso como se trabaja con un solo ordenador se usa una red local y no es necesario asignar una IP, mientras que los comandos "alias" son para usar comandos rápidos. Esto quiere decir que si en la terminal se escribe *"$ cw"* se ejecutará lo mismo que si se hubiera escrito *"$ cd ~/catkin_ws"* (utilizado para acceder al directorio "catkin_ws").

4.3 PRIMERAS PRUEBAS DE FUNCIONAMIENTO

Cuando se ha instalado y configurado ROS, se procede a realizar varias pruebas de funcionamiento para comprobar el correcto desempeño de todos los programas y herramientas. Estas pruebas se dividen en tres: La primera es la más sencilla, y consiste en comprobar si la instalación y la configuración se ha realizado correctamente. Para ello se ejecuta el siguiente comando en una nueva terminal.

$ roscore

```
roscore http://localhost:11311/
inazio@inazio-Aspire-V3-371:~$ roscore
... logging to /home/inazio/.ros/log/70374da2-cb34-11e9-86da-acb57d1d0d2a/roslau
nch-inazio-Aspire-V3-371-3039.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://localhost:32847/
ros_comm version 1.12.14


SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.14

NODES

auto-starting new master
process[master]: started with pid [3050]
ROS_MASTER_URI=http://localhost:11311/

setting /run_id to 70374da2-cb34-11e9-86da-acb57d1d0d2a
process[rosout-1]: started with pid [3063]
started core service [/rosout]
```

Figura 4.3: Información de salida al ejecutar comando "roscore"

En la terminal de la Figura 4.3 no se podrá lanzar más información ya que se está ejecutando el proceso "roscore". Para terminar el proceso es necesario presionar "Ctrl +C". Con el comando "roscore" se activa el llamado nodo maestro, por lo que cada vez que se quiere trabajar con ROS, ya sea para abrir archivos con los comandos "rosrun", "roslaunch", o simular un robot en los diferentes entornos gráficos etc., es necesario tener una terminal abierta con este comando.

El siguiente ejemplo es el paquete "turtlesim" (paquete de nodos) proporcionado por ROS. Se comprueba el funcionamiento de la comunicación entre los nodos, mensajes y "topics", algo sumamente importante a la hora de trabajar con ROS. Si este funcionamiento fallase, se puede simular el robot sin problemas, pero no se le pueden enviar mensajes por lo tanto no se podría mover ni saber cuales son sus "topics". La prueba consiste en mostrar por pantalla la simulación del logotipo de ROS Kinetic (que en este caso es una tortuga) y controlarla con el nodo (programa) del teclado.

Para ello se siguen las siguientes instrucciones, importante destacar que se tiene que escribir cada una de ellas en una terminal diferente:

$ rosrun turtlesim turtlesim_node (ejecuta la ventana gráfica donde aparece la tortuga)

$ rosrun turtlesim turtle_teleop_key (este comando permite mover la tortuga por su interfaz con el teclado

$ rqt_graph (ejecuta el esquema de nodos situado en la parte inferior de la figura)

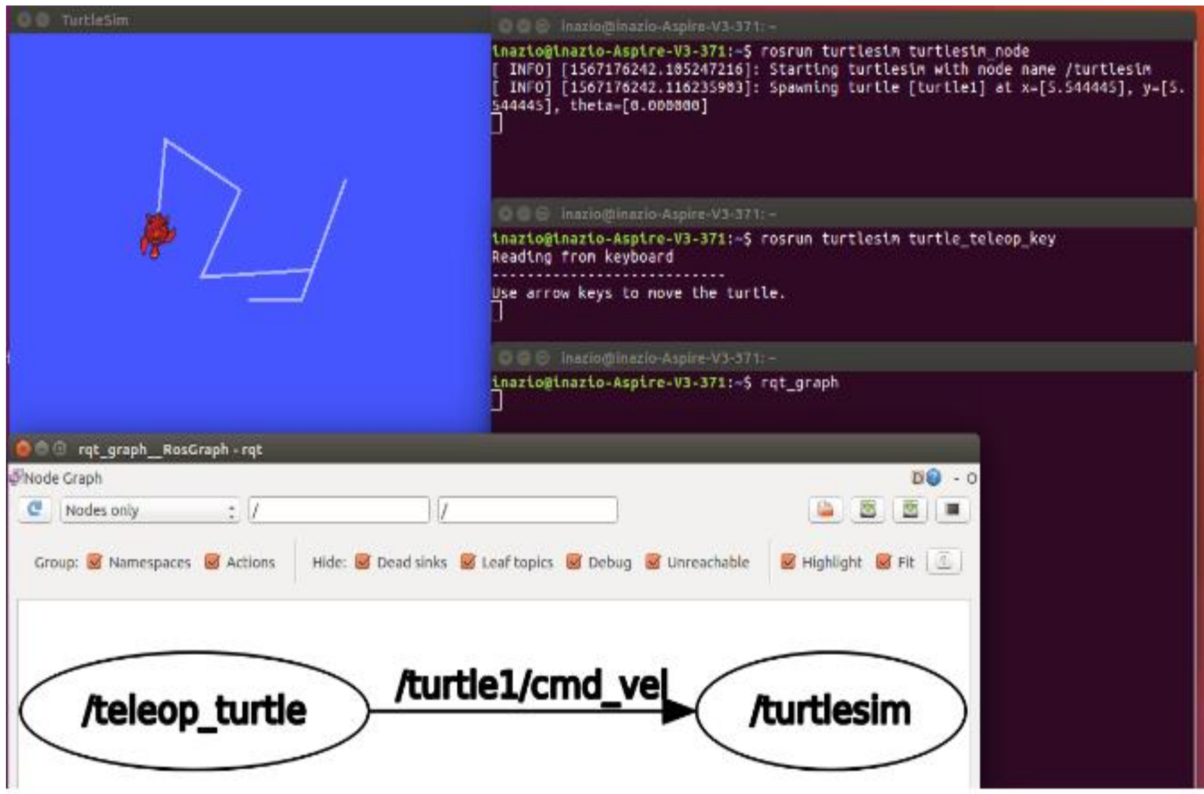


Figura 4.4: Demonstración de la segunda prueba de funcionamiento.

Cabe destacar que el último comando de la Figura 4.4 es el que nos muestra que la prueba ha sido satisfactoria, ya que muestra información sobre los nodos activos. La flecha es el “topic” y los círculos diferentes nodos entre los que se envía el mensaje. Si se observa en Figura 4.4, se dibuja una flecha desde el nodo "/ teleop_turtle" que se conecta a "/ turtlesim". Esto demuestra que los dos nodos se están ejecutando y que hay comunicación entre ellos.

Por último, se procede a la simulación del robot llamado “Turtlebot3”. Con esta prueba se verifica el correcto funcionamiento de las simulaciones, tanto en Gazebo como en RViz. Otros factores que se analizan son si funcionan los sensores de RViz (en

este caso la cámara), el poder cargar paquetes y ejecutarlos sin problemas etc.

El primer paso es cargar (mediante el comando "git clone") los paquetes en nuestro directorio "catkin" (estos paquetes son creados por las empresas "ROBOTIS" y "OPEN ROBOTICS."). Cuando se hayan copiado los cuatro paquetes se procede a su compilación. Se ejecutan los siguientes comandos:

$ cs (comando rápido para acceder a la carpeta "/catkin /src" donde se copian los paquetes)

$ git clone https://github.com/ROBOTIS-GIT/turtlebot3.git

$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git

$ git clone https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git

$ git clone https://github.com/ROBOTIS-GIT/hls_lfcd_lds_driver.git

$ cd ~/catkin_ws && catkin_make

Como se observa en la Figura 4.5, al compilar correctamente los paquetes, ya aparecen las cuatro carpetas que contienen todos los archivos para poder llevar a cabo las simulaciones.

Figura 4.5: Paquetes necesarios para llevar a cabo la simulación del "Turtlebot3".

Aunque hay varios archivos para ejecutar, en este caso se simula en Gazebo un modelo de "Turtlebot3" con cámara (modelo "waffle"), controlando sus movimientos con el teclado y se visualiza en RViz:

$ export TURTLEBOT3_MODEL=wafle (previamente a lanzar las simulaciones se escoge con qué modelo "Turtlebot3" se va a trabajar)

$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch (ejecuta la simulación de Gazebo)

$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch (se ejecuta en una nueva terminal y en ella se puede controlar mediante el teclado los movimientos del robot)

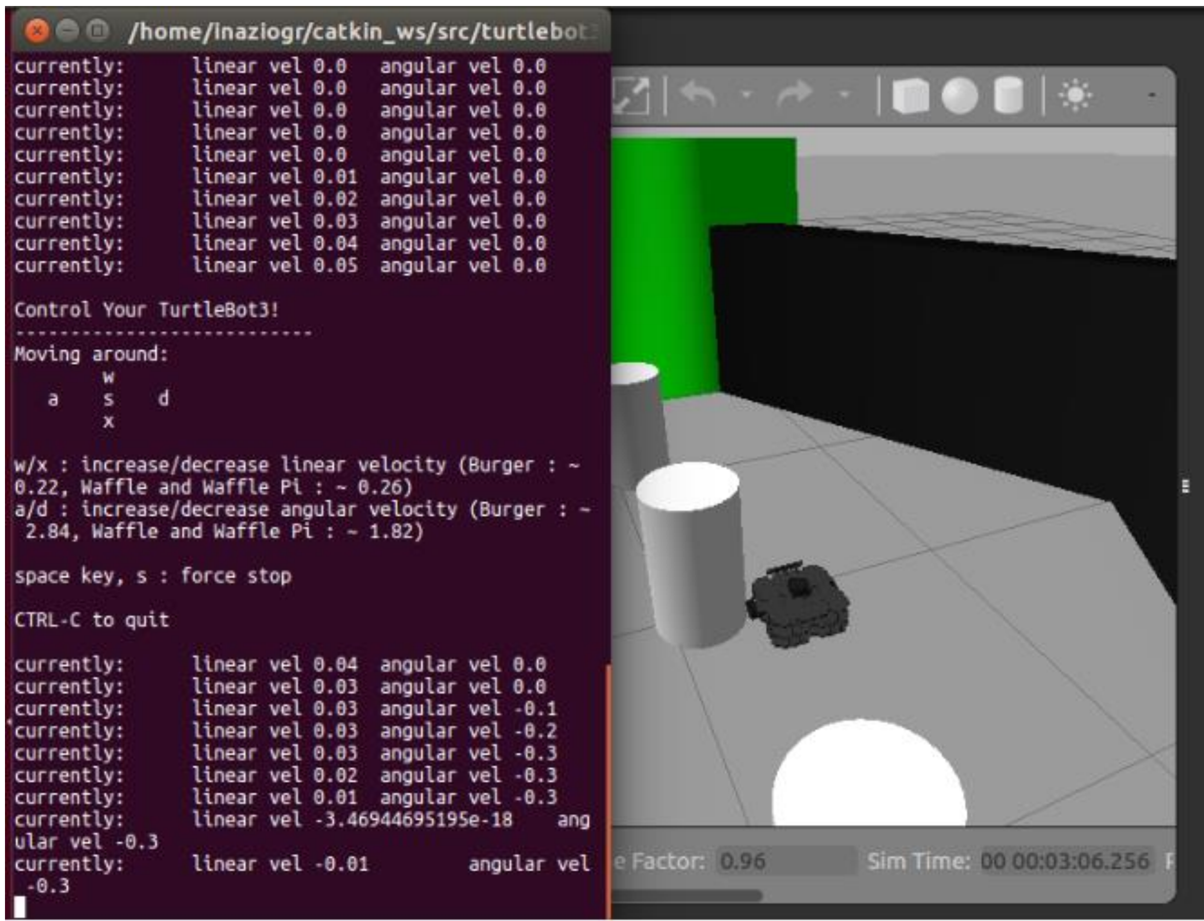


Figura 4.6: Terminal donde se controlan los movimientos de la simulación del modelo "waffle" en Gazebo.

La parte izquierda de la Figura 4.6 corresponde con la terminal con la que a través del teclado (en concreto con las letras "a", "w", "s", "x" y "d") se controla el robot. La parte derecha es la simulación en Gazebo en tiempo real.

Se abre una nueva terminal y se ejecuta el comando que vendrá a continuación, que sirve para cargar el archivo en la herramienta RViz y poder visualizarlo en esta plataforma. Se pueden activar todo tipo de sensores, pero en este caso solo se activa la cámara.

```
$ roslaunch turtlebot3_gazebo turtlebot3_gazebo_rviz.launch.
```

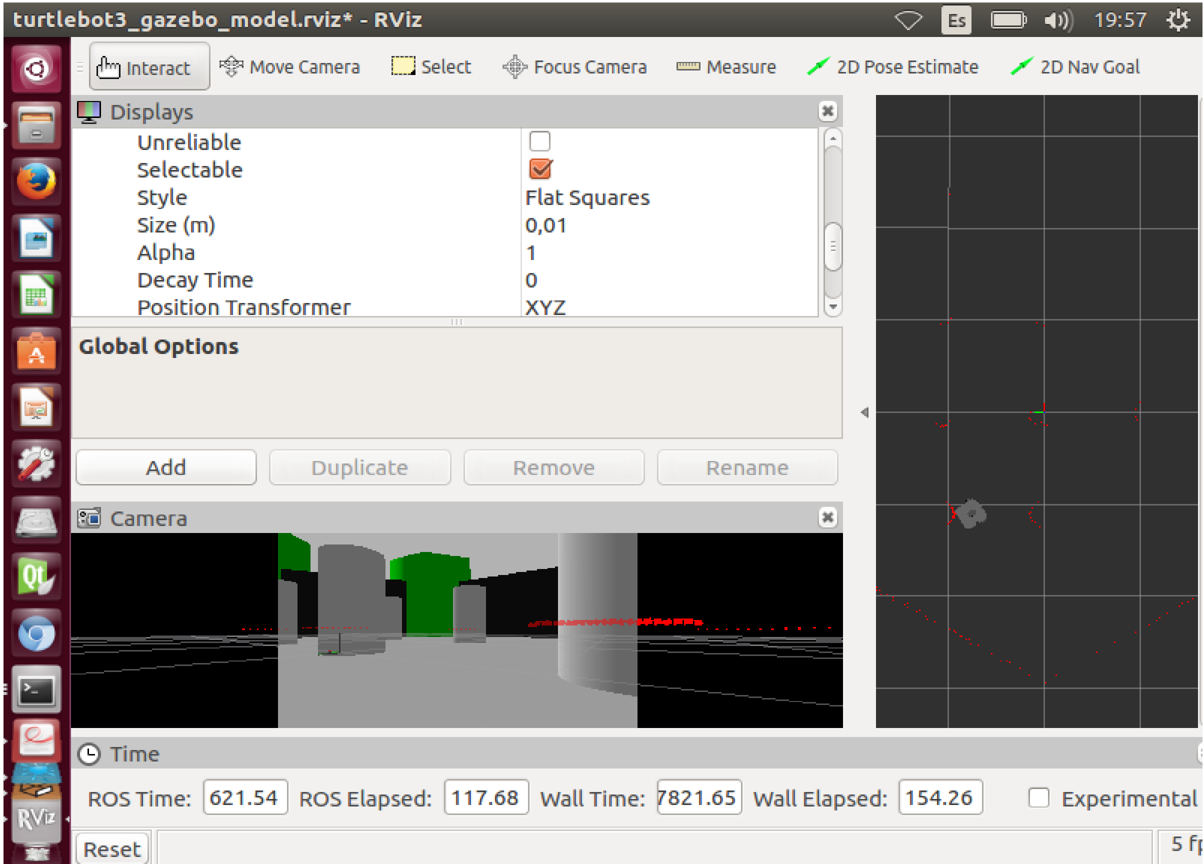


Figura 4.7: Vista del entorno RViz al activar el sensor de la cámara.

Para activar el sensor de la cámara basta con seleccionar en la pestaña de "Displays" (situada en la esquina superior izquierda) la opción "Camera". Se activa una ventana en la parte inferior izquierda, como la de la Figura 4.7, en la cual se observa la cámara del "Turtlebot3" en tiempo real. La parte de la derecha de la Figura 4.7 sirve para situar al robot en el entorno de simulación.

Si se quiere simular otro modelo de "Turtlebot3" se tiene que modificar el comando "export". Se prueba también a simular otro modelo diferente con los siguientes comandos:

$ export TURTLEBOT3_MODEL=burguer (se simula el modelo "burguer")

$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch

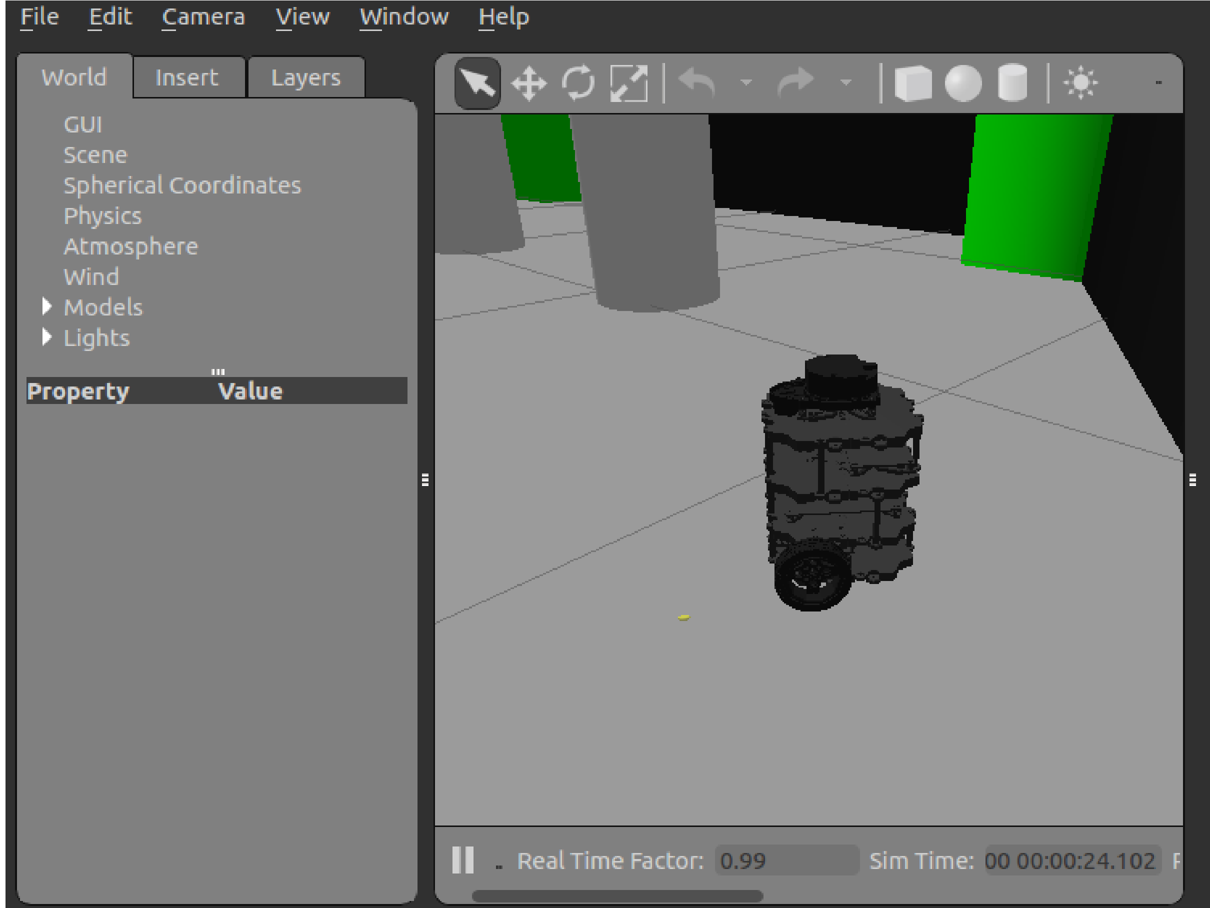


Figura 48: Simulación del modelo "Burger" en Gazebo.

En la Figura 4.8 se muestra la simulación en Gazebo del modelo "burguer". También se puede manejar con el teclado si se repiten los comandos vistos anteriormente.

Tras la realización de estas tres pruebas, se obtiene la conclusión de que tanto la instalación, configuración y funcionamiento son correctas. Es importante que todas sean satisfactorias ya que, en caso de no obtener el resultado esperado, posteriormente en las simulaciones aparecerán errores más complicados de solucionar. En el Anexo C, se han recopilado varios de los errores que podemos obtener al realizar dichas pruebas y su método de resolución.

5.Control de robots industriales ABB

En este quinto Capítulo se ponen en marcha las primeras simulaciones de robots industriales, que servirán como modelo para ejecutar diferentes movimientos. Éstos, son fabricados por una de las empresas punteras en automatización industrial, como es ABB (acrónimo de Asea Brown Boveri), que ha instalado más de 300.000 robots en todo el mundo. Se describen tres métodos para planificar movimientos, incluyendo en uno de ellos (MoveIt) toda la configuración necesaria para su simulación tanto en Gazebo como en RVIz. Por último, se aplica uno de los métodos para el movimiento simultáneo de dos robots en un mismo entorno de Gazebo.

5.1 ROBOTS ABB

ABB proporciona una completa gama de robots industriales, tanto articulados, como Scara o colaborativos, para ayudar a los fabricantes a mejorar la productividad y la calidad del producto. Para poder programar robots ABB, se ha diseñado un software de simulación llamado RobotStudio (ejemplo en la Figura 5.1), que permite efectuar la programación del robot en un ordenador en la oficina sin interrumpir la producción. En los siguientes apartados se simularán robots de ABB sin necesidad de usar RobotStudio, sino gracias a ROS [15].

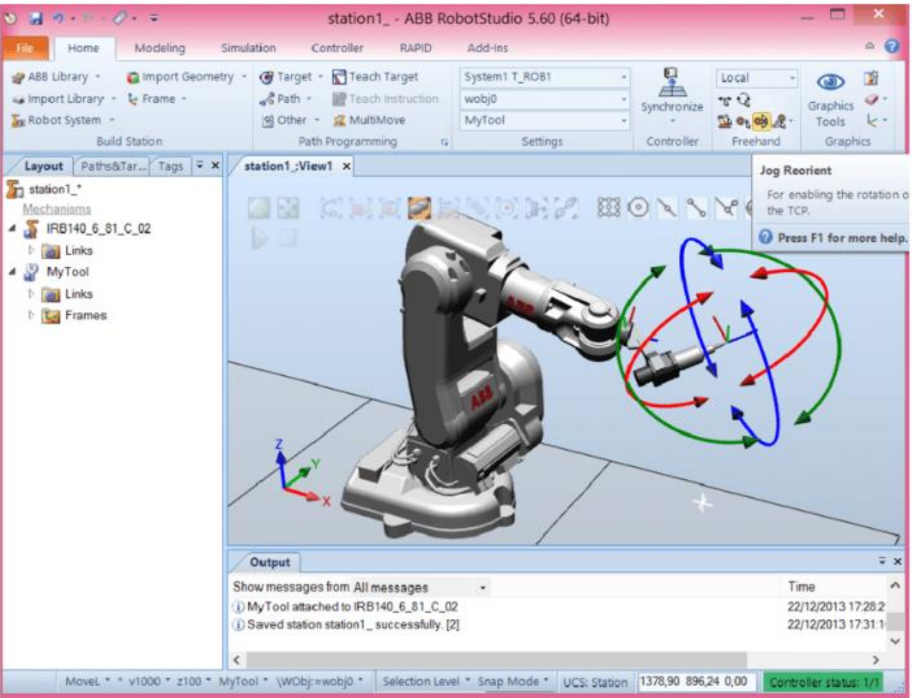


Figura 5.1: Simulación de un robot ABB en el entorno de RobotStudio [15].

De entre todos los innumerables modelos se va a trabajar con el modelo IRB 120, cuyas características quedan reflejadas en el Anexo A. Se trata de un brazo robot de pequeñas dimensiones y bajo peso capaz de llevar a cabo múltiples tareas, como pueden ser: paletizado, ensamblado de electrónica de consumo, "pick and place", etc. Se ha elegido simular este modelo ya que es el más utilizado por las empresas debido a su polivalencia y su escaso tamaño (es el robot articulado más pequeño que fabrica ABB).

El IRB 120 cuenta con un modelo muy similar que es el IRB 120T, este es referente para aplicaciones pick & place rápidas ya que da mejoras en los tiempos de ciclo de hasta un 25%. A la hora de simularlos será indiferente simular cualquiera de los dos ya que las diferencias en sus códigos URDF son casi inapreciables.

Es fácil encontrar numerosos paquetes para ROS que contienen simulaciones de robots de ABB pero se ha elegido el siguiente: *"https://github.com/ros-industrial/abb_experimental"*. Uno de los múltiples modelos que contiene es el IRB 120, junto a su variante IRB 120T. Se ejecutan los siguientes comandos para copiar el paquete y compilarlo:

```
$ cs (comando rápido para acceder a la carpeta "/catkin_ws/src")
$ git clone https://github.com/ros-industrial/abb_experimental.git
$cd ~/catkin_ws && catkin_make
```

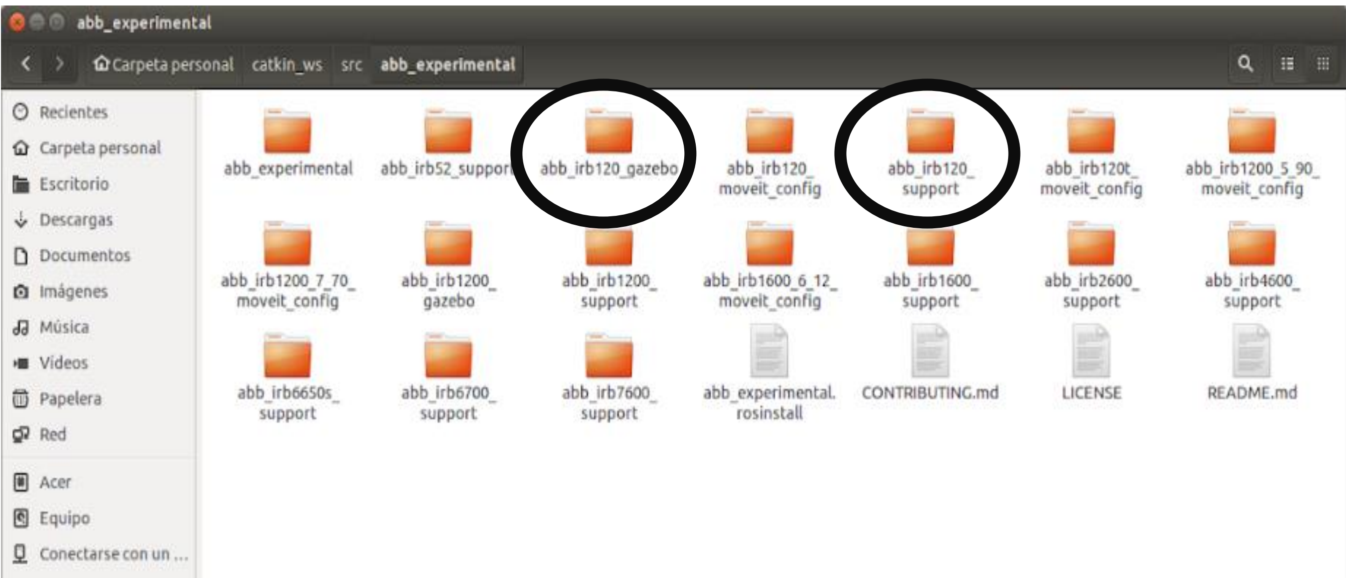


Figura 5.2: Contenido del paquete "abb_experimental" tras su compilación, destacando las dos carpetas con las que se va a trabajar.

Como se aprecia en la figura 5.2 hay numerosas carpetas, pero se van a trabajar con las dos que están destacadas que son: “abb_irb120_gazebo” y “abb_irb120_support”. La primera de ellas contiene todos los códigos para lanzar los archivos y simular el IRB 120 en gazebo. La segunda carpeta está compuesta por los archivos que incluyen toda la información de cómo se ha diseñado el robot, es decir, los archivos URDF donde se describen los “links”, “joints”, masa, geometría etc.

Con los comandos que se escriben a continuación se simula el modelo IRB 120 en Gazebo. Aparte, como se puede apreciar en la esquina inferior izquierda de la Figura 5.3, se muestran todos los “topics” del robot:

$ roslaunch abb_irb120_gazebo irb120_3_58_gazebo.launch

$ rostopic list (necesario ejecutar en otra terminal para ver los “topics” que están en activo)

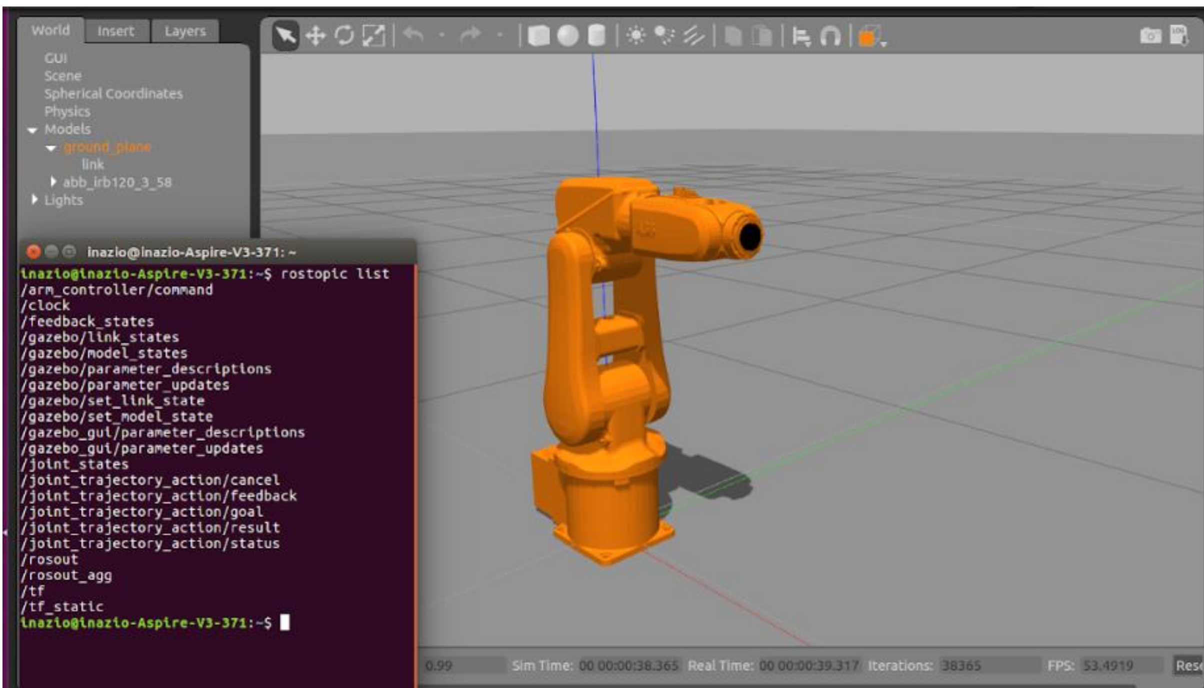


Figura 5.3: Simulación del modelo IRB120 en Gazebo, mostrando todos los “topics” en activo.

5.2 CONFIGURACIÓN DEL ROBOT IRB 120 CON “MOVEIT”

En este apartado se utiliza el asistente de “Moveit” para configurar un modelo de robot. Se añaden los diferentes grupos de articulaciones, las posiciones del robot que se quieran etc. Esta configuración se realiza para que, en el siguiente capítulo, sea posible ejecutar movimientos del robot simulando el resultado en Gazebo. Lo primero se abre el asistente de “Moveit”:

```
$ roslaunch moveit_setup_assistant setup_assistant.
```

Como lo que se quiere es crear una nueva configuración, se selecciona la opción de "Create new moveit configuration package", y dentro de la carpeta de "abb_irb120_support" se busca el archivo correspondiente. Toda la información del modelo IRB 120 está en el archivo "irb120_3_58_macro.xacro" pero "Moveit" no puede abrir archivos ".xacro". En este caso ya está creado el archivo URDF correspondiente si no, se tendría que ejecutar el siguiente comando:

```
$ rosrun xacro xacro.py NOMBRE_DEL_ARCHIVO.xacro > NOMBRE_DEL_NUEVO_
ARCHIVO.urdf
```

Se selecciona la opción "Load Files" y se carga el robot, como se visualiza en la Figura 5.4. A la izquierda, se muestran todas las selecciones que posee el asistente y a la derecha está situada la simulación del robot.

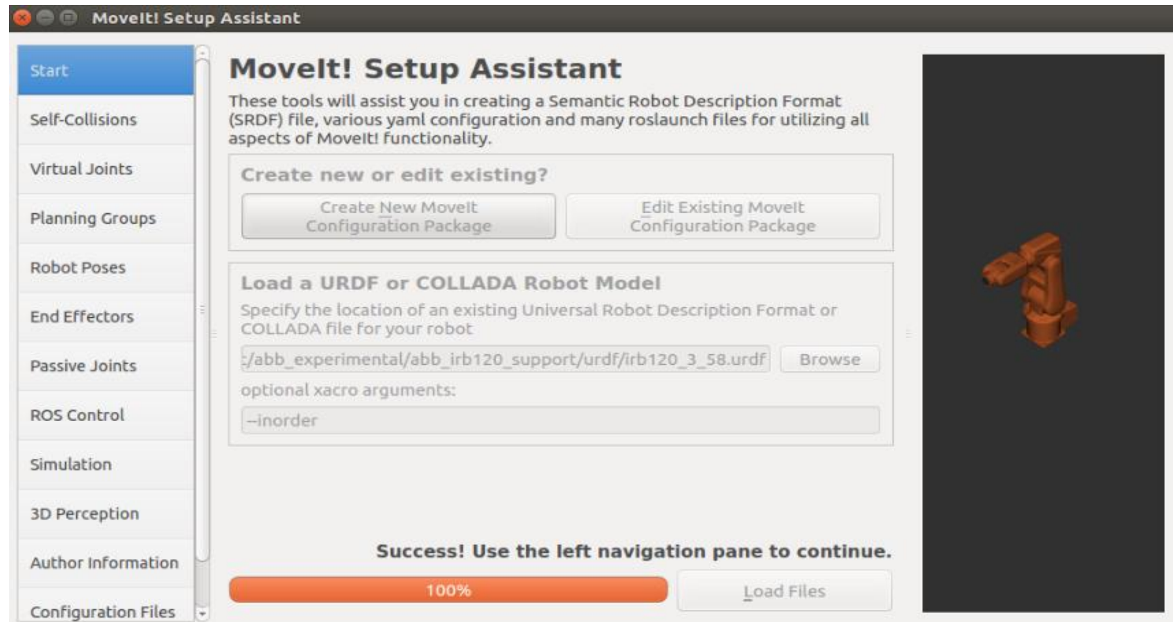


Figura 5.4: Asistente de "MoveIt" tras elegir el archivo para crear su nueva configuración.

- **"Self-Collisions":** Esta primera opción permite definir la densidad de muestreo necesaria para construir la "matriz de auto-colisión". Se puede determinar el rango de colisiones entre los enlaces que componen el robot, que influye en el tiempo de procesamiento de planificación de movimiento. Cuanto mayor es la densidad de muestreo, se requiere más cálculo para evitar la colisión entre enlaces en varias poses del robot. El valor que se introduce es el máximo (10,000). Un ejemplo de la matriz una vez generada se muestra en la Figura 5.5.

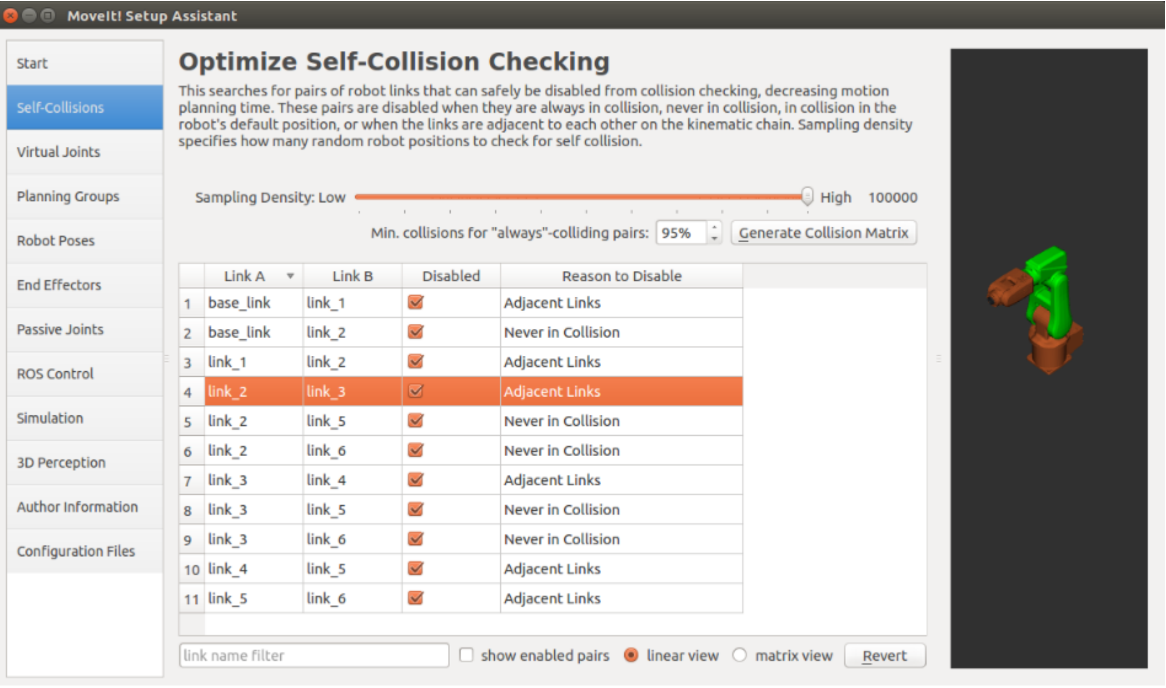


Figura 5.5: Captura del asistente tras generar la matriz de colisión.

- **"Virtual Joints":** Proporciona la opción de crear una unión virtual entre la base del robot y el sistema de coordenadas de referencia. La unión representa el movimiento de la base del robot en un plano. Se crea una unión virtual entre la base ("base_link") y el sistema de referencia ("world"), con la opción de ser fija ("fixed"), como se plasma en la Figura 5.6.

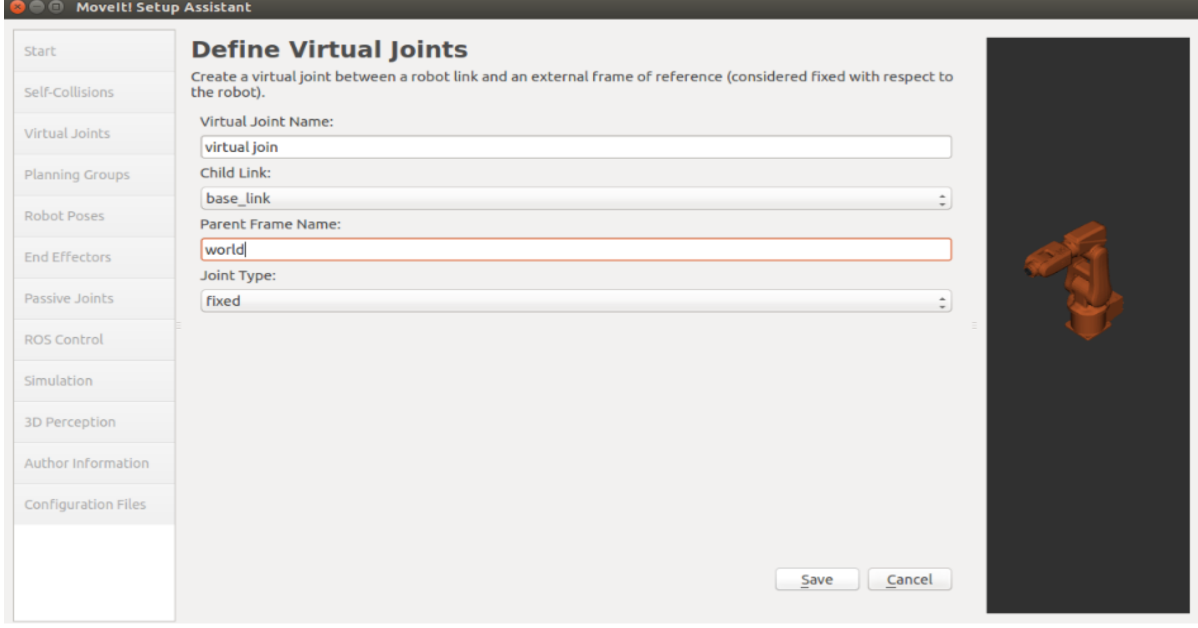


Figura 5.6: Pestaña de "Virtual Joint".

- **"Planning Groups":** Los grupos de planificación se utilizan para describir semánticamente diferentes partes del robot, como definir las partes del brazo o de cualquier herramienta que utilice. Al estar compuesto por un brazo, se crea un solo grupo llamado "arm" en el cual se añaden los 6 "joints" del robot. Se eligen las opciones tal y como se muestra en la Figura 5.7. En las opciones numéricas se dejan las opciones predeterminadas, mientras que las otras dos restantes se configuran para conectarse a la simulación de Gazebo como se describe en el Capítulo 5.3.3. El resultado definitivo se visualiza en la Figura 5.8.

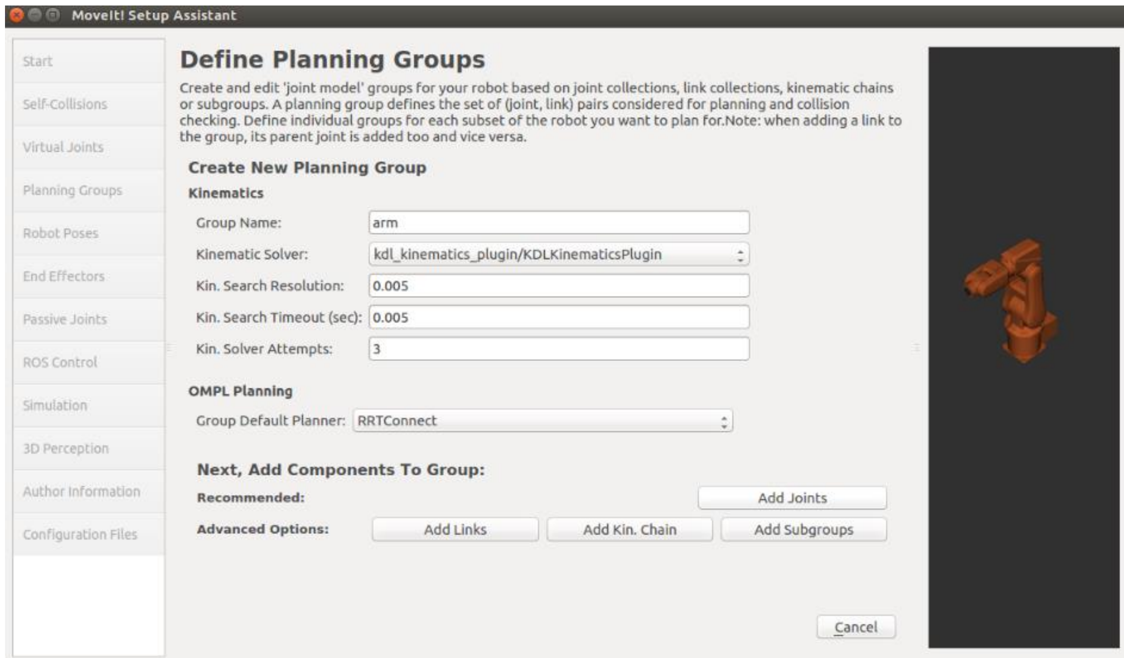


Figura 5.7: Opciones escogidas en el apartado "Planning groups".

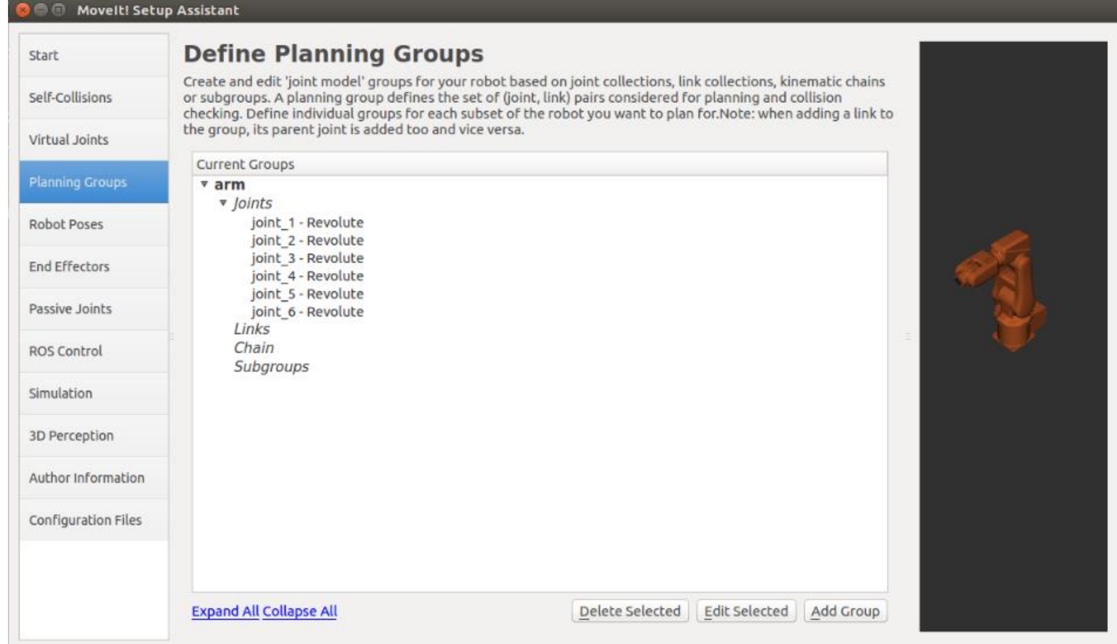


Figura 5.8: Grupo "arm" creado con sus respectivos joints.

- **"Robot Poses":** En esta pestaña se definen las posiciones del robot. Es muy útil crear la postura "zero_pose" donde todos los "joints" están en su posición inicial. Cuando se simule, no será necesario moverlo manualmente para que vuelva a su posición de reposo, con seleccionar la posición "zero_pose", el robot se moverá a su posición de partida. Aparte, se crea otra posición llamada "aleatoria", definida en la Figura 5.9.

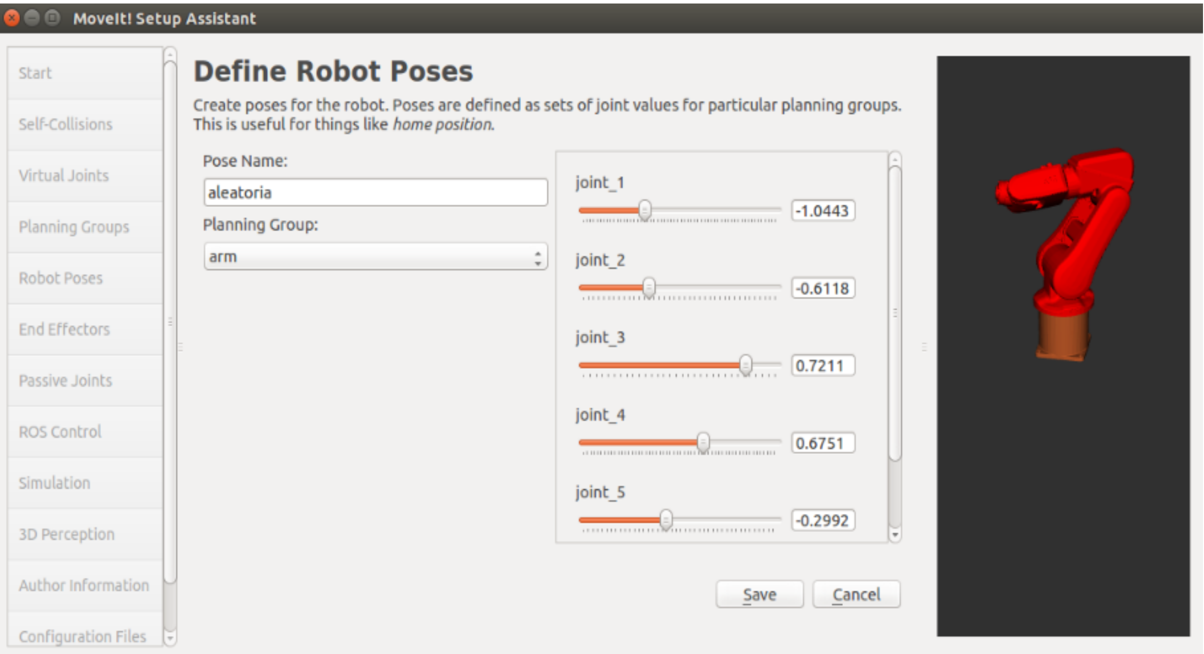


Figura 5.9: Creación de una posición aleatoria del robot.

- **"End Effectors":** Sirve para añadir al "Planning Groups" herramientas, "grippers" etc. Esta opción no se modifica, debido a que el modelo de fábrica del IRB 120 no contiene ninguna herramienta.

- **"Passive joints":** Permite excluir a cualquier "joint" o junta de la planificación de movimientos, pero en ese caso no se quiere excluir ninguna por lo que se pasa a la siguiente pestaña.

- **"Simulation y 3D Perception":** La opción "Simulation" permite generar un archivo URDF para simular al robot. "3D Perception" es la encargada de añadir o configurar los diferentes sensores que tiene el robot, como pueden ser cámaras, sensores de movimiento etc. (al modelo IRB 120 no se le añade ningún sensor que se pueda configurar en esta pestaña).

- **"ROS Control":** Conjunto de paquetes que incluyen interfaces de hardware, administradores de controlador y transmisiones. La pestaña Control de ROS se puede usar para generar automáticamente controladores simulados para accionar las

uniones de su robot. Esto nos permitirá proporcionar las interfaces de ROS correctas. Se añade un controlador al brazo del tipo “position_controllers/JointPositionController”, que permite a través de comandos poder mover el robot a cualquier posición. Se muestra la configuración en la Figura 5.10

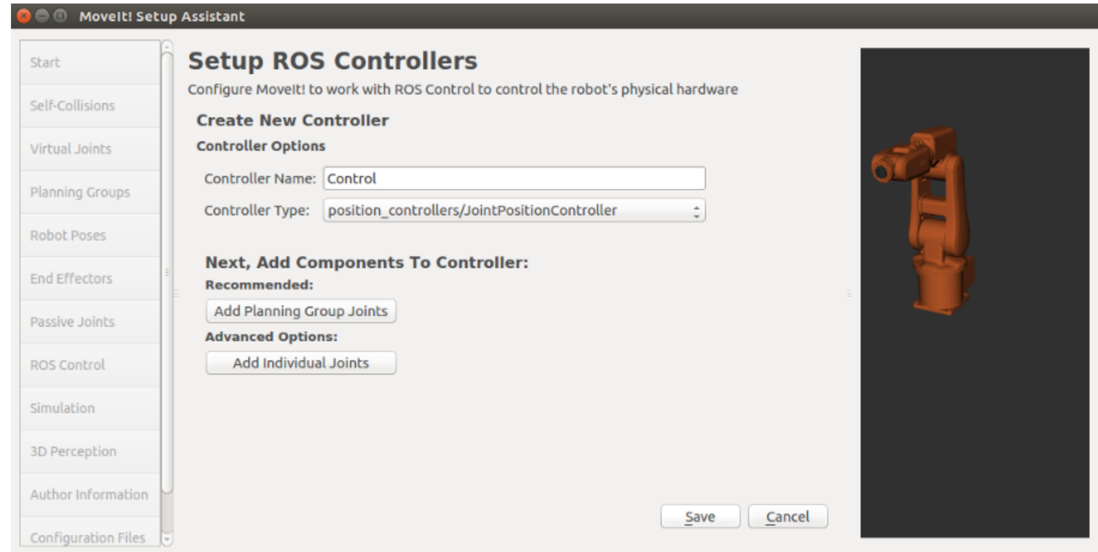


Figura 5.10: Pestaña “ROS Control” para definir un controlador al robot.

- **“Author Information” y “Configuration Files”:** Es necesario dar un nombre y un correo, para guardar los datos en el apartado de “Author Information”, y escoger una carpeta donde guardar el paquete de “Moveit” (se crea una llamada “IRB_120_MoveIt”). En el apartado donde se puede elegir que archivos se van a generar, se seleccionan todos. Por último, se genera el paquete haciendo “click” en “Generate Package”, si no hay problemas aparece un mensaje como el de Figura 5.11

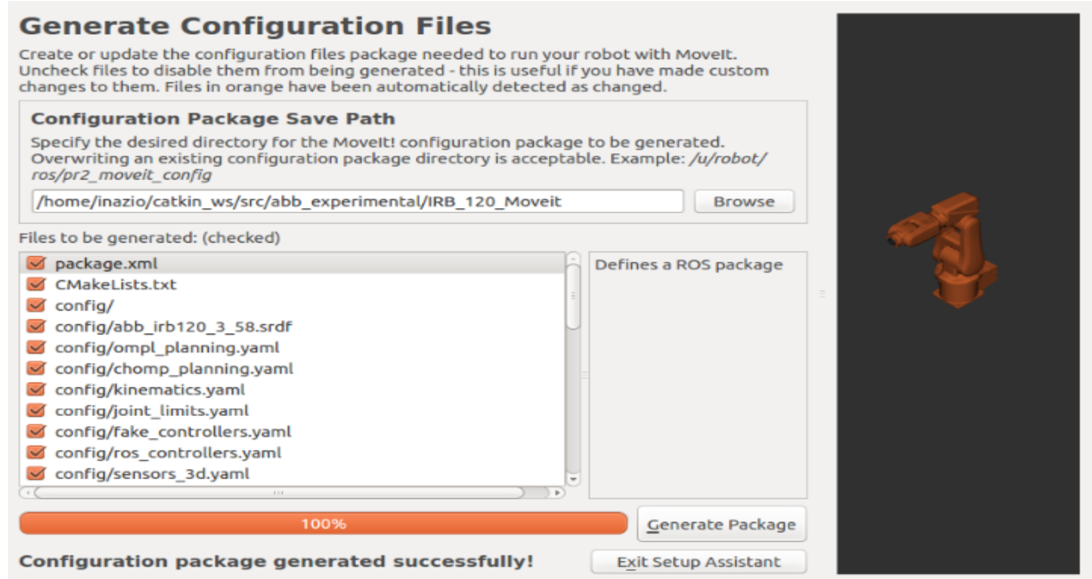


Figura 5.11: Último paso del asistente al generar el paquete de configuración.

5.3 PLANIFICACIÓN DE MOVIMIENTOS

Para poder realizar cualquier tarea de manipulación, es importante conocer los diferentes métodos de movimiento de un robot, en una simulación en Gazebo. En este apartado se describen tres métodos diferentes:

5.3.1 Movimientos por comandos

El primer procedimiento, es el más sencillo y sobre todo el más rápido de ejecutar. Para ello, se tiene que definir uno de los "topics" del robot para que se pueda mover con alguna de las siguientes configuraciones: "Effort Joint Interface", "Velocity Joint Interface" o "Position Joint Interface" (se amplía la información sobre estas configuraciones en el Anexo C). Tras buscar en la configuración del robot, se encuentra el "topic" que se quiere ir publicando, en este caso es "arm_controller", y es un controlador de posición ("position_controller").

Para obtener más información de este "topic" se simula el robot y se ejecutan los comandos de la Figura 5.12.

$ roslaunch abb_irb120_gazebo irb120_3_58_gazebo.launch

$ rostopic list (devuelve todos los "topics" en activo)

$ rostopic type /arm_controller/command (devuelve de qué tipo es el "topic")

-El siguiente comando también devuelve de qué tipo es.

$ rostopic info /arm_controller/command

```
inazio@inazio-Aspire-V3-371: ~
/tf_static
inazio@inazio-Aspire-V3-371:~$ rostopic list
/arm_controller/command
/clock
/feedback_states
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/gazebo_gui/parameter_descriptions
/gazebo_gui/parameter_updates
/joint_states
/joint_trajectory_action/cancel
/joint_trajectory_action/feedback
/joint_trajectory_action/goal
/joint_trajectory_action/result
/joint_trajectory_action/status
/rosout
/rosout_agg
/tf
/tf_static
inazio@inazio-Aspire-V3-371:~$
```

```
inazio@inazio-Aspire-V3-371: ~
inazio@inazio-Aspire-V3-371:~$ rostopic info /arm_controller/command
Type: trajectory_msgs/JointTrajectory

Publishers: None

Subscribers:
 * /gazebo (http://localhost:35269/)


inazio@inazio-Aspire-V3-371:~$ rostopic type /arm_controller/command
trajectory_msgs/JointTrajectory
inazio@inazio-Aspire-V3-371:~$
```

Figura 5.12: Terminales donde se han ejecutado los comandos necesarios para visualizar los "topics" y conocer el tipo de uno en concreto.

Como se observa en la Figura 5.13, el "topic" es del tipo "trajectory_msgs/JointTrayectory, para poder modificar las propiedades de este es necesario introducir el comando de la siguiente fotografía, con el cual se moverá el robot.

```
inazio@inazio-Aspire-V3-371:~$ rostopic pub /arm_controller/command trajectory_msgs/JointTrajectory "header:
>   seq: 0
>   stamp:
>     secs: 0
>     nsecs: 0
>   frame_id: ''
> joint_names:
> - ['joint_1',joint_2',joint_3',joint_4',joint_5',joint_6']
> points:
> - positions: [0.0, -1.0, 0.0, 2.0, 0.0, 0.0]
>   velocities: [0]
>   accelerations: [0]
>   effort: [0]
>   time_from_start: {secs: 2, nsecs: 0}"
publishing and latching message. Press ctrl-C to terminate
```

Figura 5.13: Terminal con el comando que se utiliza para mover al robot.

Se escriben todos los "joints" del robot y el desplazamiento de cada uno de ellos en el apartado "positions". En este caso, se mueve el "joint_2" y el "joint_4" (se visualiza el robot tras ejecutar el movimiento en la Figura 5.15) También se puede introducir valores de velocidad, aceleración etc., pero al no ser necesarios se dejan con su valor predeterminado.

Cuando se lanza el programa, si todo es correcto aparece una línea en la terminal (situada en la parte inferior de la Figura 5.13) que indica que se está publicando el mensaje, por lo tanto, modificando el "topic" y moviendo el robot.

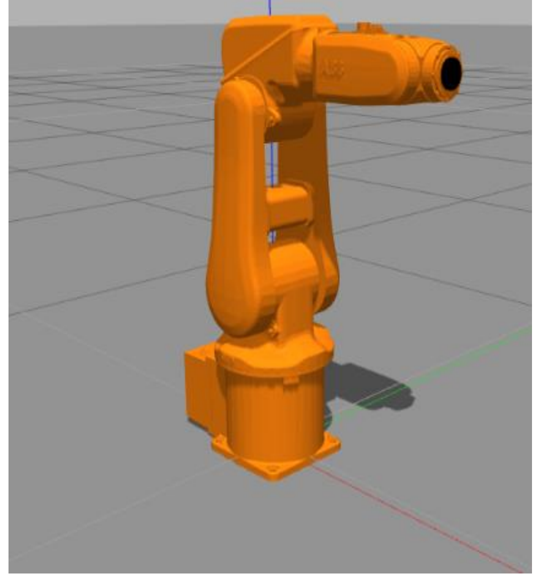


Figura 5.14: Simulación en Gazebo del robot en su posición de partida

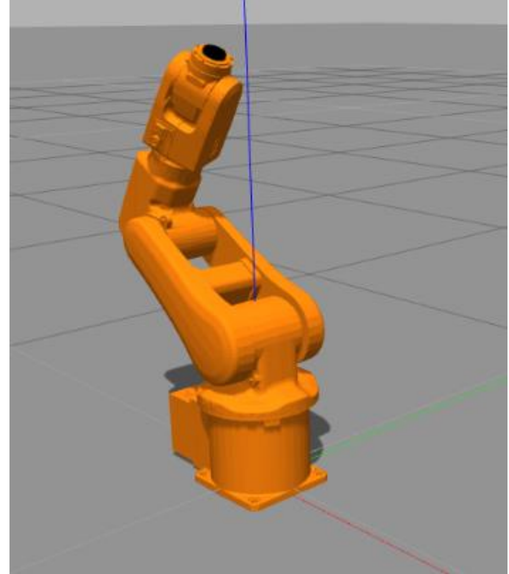


Figura 5.15: Simulación del robot una vez se ha ejecutado el comando de la Figura 5.13.

Para saber si realmente si ha modificado la posición del robot, se utiliza el siguiente comando, que devuelve el valor del "topic", es decir te devuelve el valor de la posición del robot.

```
$ rostopic echo /arm_controller/command
```

5.3.2 Programación de movimiento en Python

El anterior método es eficaz si queremos mover el robot a una sola posición con rapidez. Normalmente un robot manipulador repite continuamente varios movimientos, por lo que utilizar este método sería bastante trabajoso.

Se ha implementado un programa en "Python", adaptando lo visto en el apartado anterior, para que el robot sea capaz de ejecutar varios movimientos seguidos lanzando una sola orden en la terminal (se muestra en la Figura 5.16).

```python
#!/usr/bin/python
from trajectory_msgs.msg import JointTrajectory
from std_msgs.msg import Header
from trajectory_msgs.msg import JointTrajectoryPoint
import rospy

def main():

    rospy.init_node('send_joints')
    pub = rospy.Publisher('/arm_controller/command',
                          JointTrajectory,
                          queue_size=10)
    # Se crea el mensaje
    traj = JointTrajectory()
    traj.header = Header()
    traj.joint_names = ['joint_1', 'joint_2',
                        'joint_3', 'joint_4', 'joint_5',
                        'joint_6']

    # Bucle para modificar la posicion de los 'joints'
    rate = rospy.Rate(10)
    for x in range(0, 10000):
        traj.header.stamp = rospy.Time.now()
        pts = JointTrajectoryPoint()
        pts.positions = [0.0, 1.0, 0.0, 0.0, 0.0, 0.0]
        pts.time_from_start = rospy.Duration(1.0)
        traj.points = []
        traj.points.append(pts)

        # Se publica el mensaje
        pub.publish(traj)


if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        print ("Program interrupted before completion")
```

Figura 5.16: Captura del archivo escrito en "python".

Para ejecutar el código es necesario simular el robot en Gazebo y escribir el siguiente comando en una nueva terminal:

```
$ rosrun abb_irb120_gazebo abb_python.py
```

5.3.3 Movimientos con "MoveIt" y RViz

La utilidad de lo visto en el Capítulo 5.2 se ve reflejada en este apartado. Cuando se crea el paquete de "MoveIt" se generan varios archivos, pero el único que se va a utilizar es "Moveit_planning_execution_gazebo.launch"

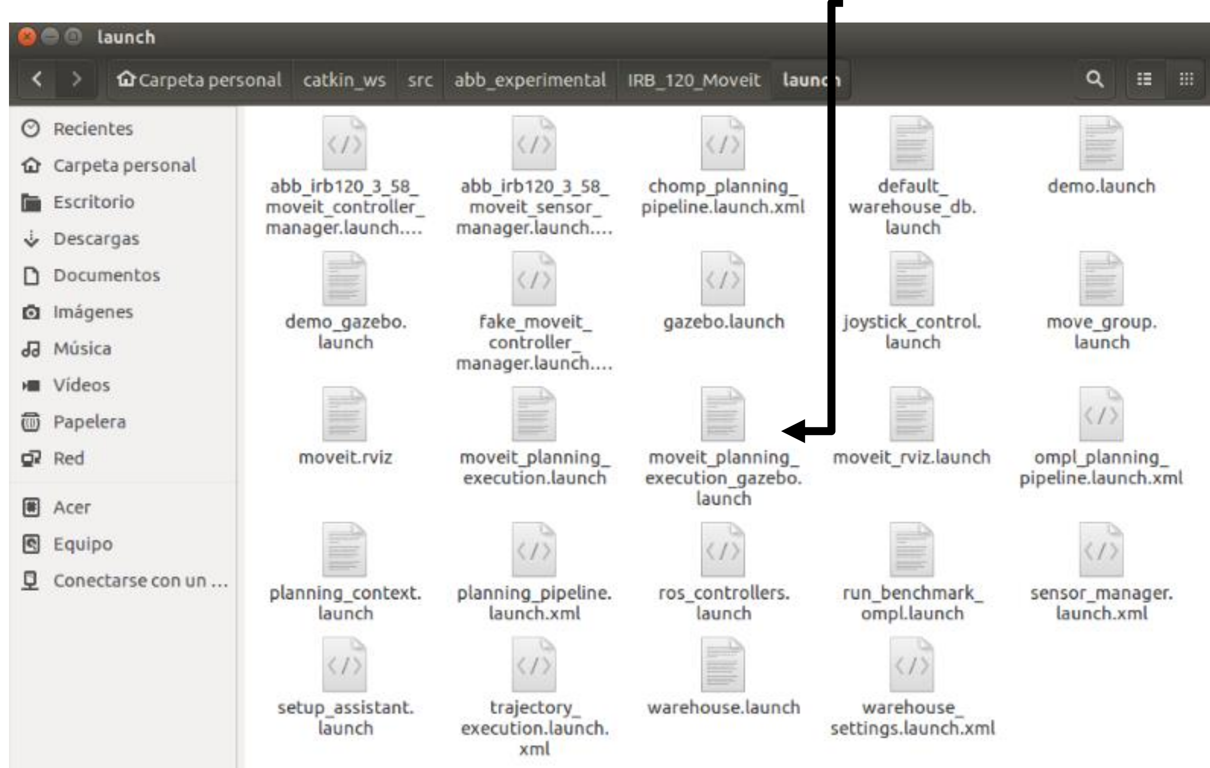


Figura 5.17: Contenido del paquete creado con el asistente de "MoveIt".

En una terminal se lanza la simulación en Gazebo y en otra diferente el archivo mencionado anteriormente, que inicia la herramienta RViz, ejecutando los siguientes comandos:

```
$ roslaunch abb_irb120_gazebo irb120_3_58_gazebo.launch
$ roslaunch IRB_120_Moveit Moveit_planning_execution_gazebo.launch
```

Una vez activas ambas herramientas, lo primero es conectarse al robot. En RViz aparece una opción en la esquina inferior izquierda llamada “MotionPlanning”. Ésta, contiene varias pestañas y en la primera de todas (“Context”) se selecciona la opción que se ha elegido anteriormente en el asistente en la Figura 5.7. (en “MoveIt” corresponde con “RRTConnectkConfigDefault”, como muestra la figura 5.18.)

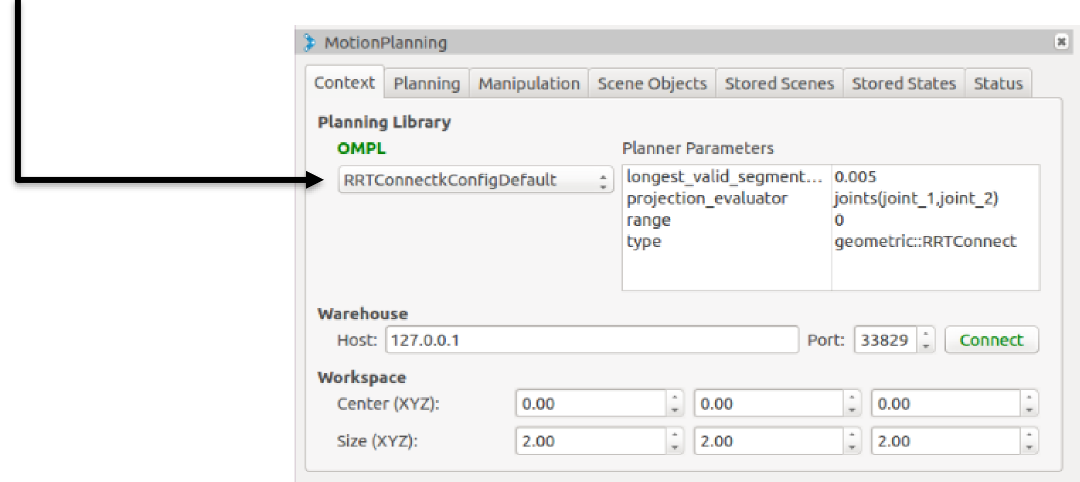


Figura 5.18: Recorte de la opción “Motión Planning”.

Otra configuración realizada previamente es el “Interactive Marker” o esfera utilizada para mover el robot. En la Figura 5.19, se observa que podemos variar ciertos parámetros de este marcador interactivo, por ejemplo, su tamaño en “Interactive Marker Size”. Pero lo importante es que se puede cambiar de “Planning group”. En el caso de que el robot tuviera varios brazos y se hubieran creado más de un grupo, este marcador podría ir cambiando de “Planning group” para poder mover cualquiera de los brazos.

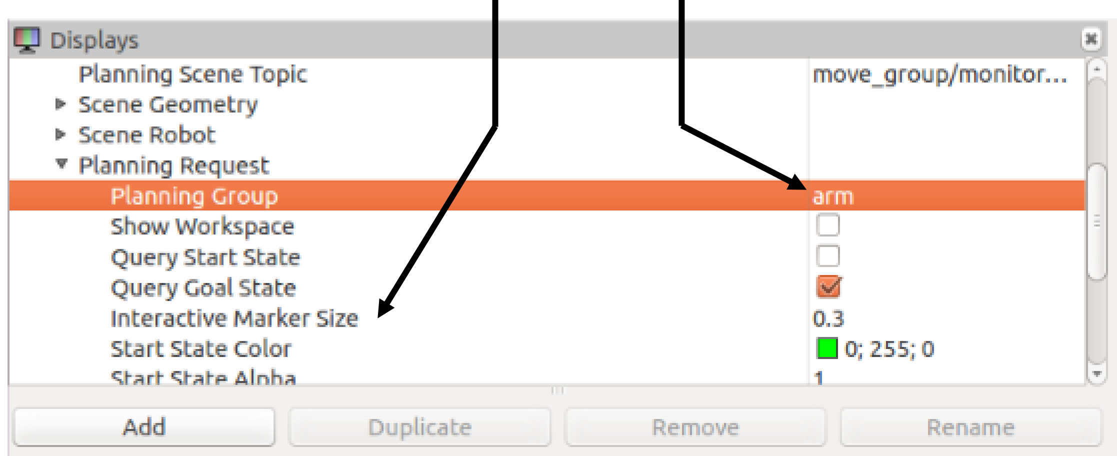


Figura 5.19: Opción “Displays” de la simulación en RViz.

Para mover el robot, se hace "click" en el "Interactive Marker" y con el ratón se desplaza hasta la posición deseada. Si se quiere dar una posición precisa, basta con seleccionar en la pestaña "Query" la posición que se ha definido previamente en el asistente de "MoveIt". En la simulación de la Figura 5.20, se tienen definidas las dos posiciones vistas anteriormente, que son las de "zero_pose" y "aleatoria"

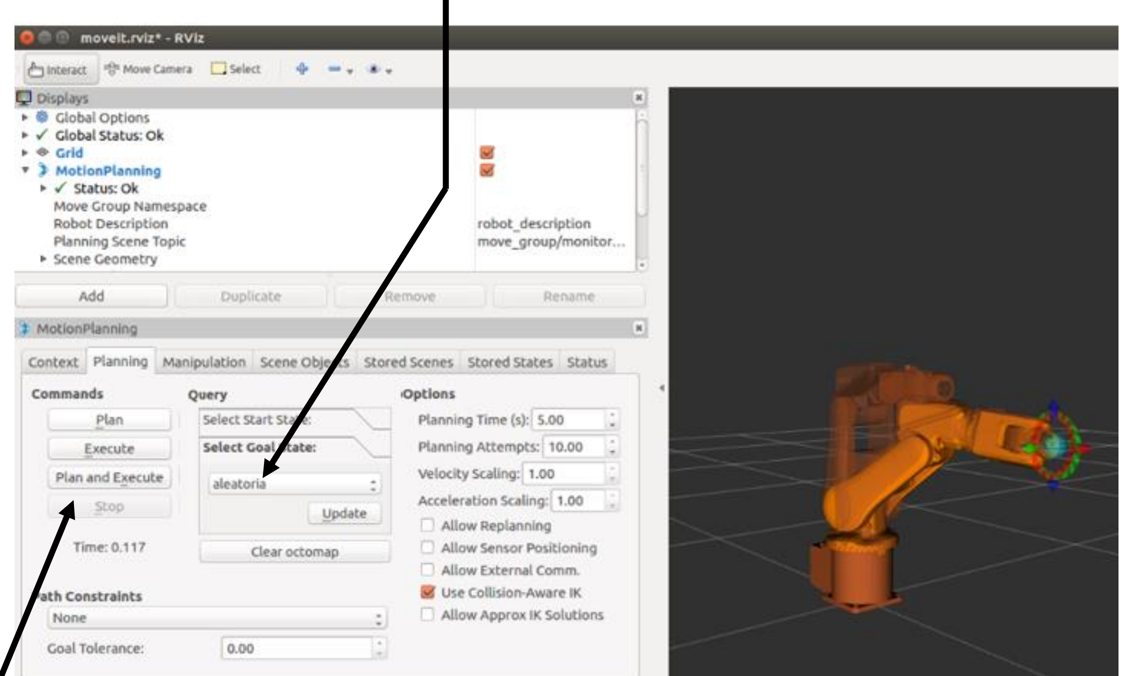


Figura 5.20: Visualización de Rviz tras mover el robot con el marcador interactivo.

Para enviar la acción al robot simulado en Gazebo, se selecciona la opción "Plan and Execute" y el resultado es el de la Figura 5.21.

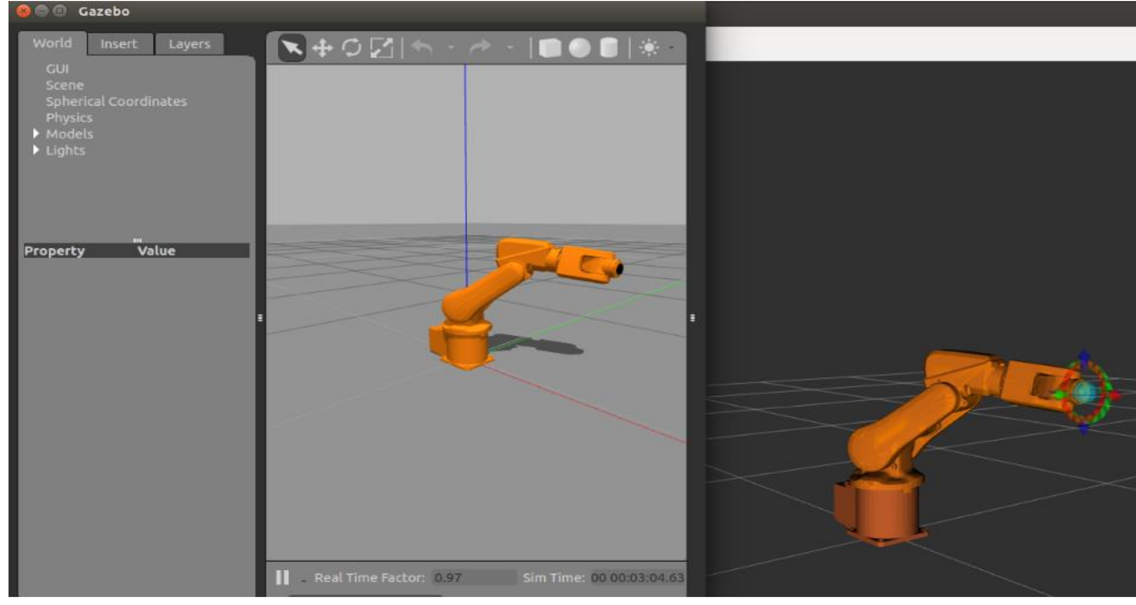


Figura 5.21: Captura de las simulaciones de Gazebo (izquierda) y RViz (derecha).

5.4 SIMULACIÓN MULTIROBOT

Para cualquier industria, el hecho de tener procesos automatizados les ofrece numerosos beneficios, sobre todo en cuanto a la producción, como pueden ser menores costes de fabricación y más eficiencia de resolución (rapidez y calidad). Por lo que, cada vez más, se sustituyen tareas que se hacían a mano por uno o varios operarios por células de trabajo con más de un robot. Por dicho motivo, en este apartado se van a simular varios robots del mismo modelo (IRB120) y se ejecutaran diferentes movimientos para cada uno.

Para conseguir esta simulación se modifica el archivo "*irb120_3_58_gazebo.launch",* utilizado anteriormente para lanzar la simulación de un solo robot. Existen varios métodos para poder controlar más de un robot en un mismo entorno. El primero consiste en duplicar los archivos URDF y los archivos que contienen los controladores del robot (modificando los nombres que se les adjudica a los "joints" y "links"). Este método puede ser más complejo a la hora de ejecutar movimientos, ya que se tendrían dos robots cuyos nombres de "joints" y "links" son distintos y puede llevar a equivocación al escribir el código.

El segundo es el método utilizado ya que más sencillo y consiste en dividir en dos subgrupos al robot:

```
  <arg name="first_robot_ip" value="robot_description"/>
  <arg name="second_robot_ip" value="robot_description"/>

  <group ns="first">

  <param name="robot_description" command="$(find xacro)/xacro --inorder '$(find
abb_irb120_gazebo)/urdf/irb120_3_58.xacro'" />

  <node name="abb_irb120_spawn" pkg="gazebo_ros" type="spawn_model" output="screen" args="-
urdf -param robot_description -x 1.0 -y 0.0 -z 0.0 -R 0.0 -P 0.0 -Y 0.0 -model
abb_irb120_3_58" respawn="false" />

  <rosparam file="$(find abb_irb120_gazebo)/config/joint_state_controller.yaml"
command="load" />

      <arg name="robot_ip" value="$(arg left_robot_ip)" />

   <!-- load the arm controller -->
  <rosparam file="$(find abb_irb120_gazebo)/config/irb120_3_58_arm_controller.yaml"
command="load" />
  <node name="abb_irb120_controller_spawner" pkg="controller_manager"
type="controller_manager" args="spawn arm_controller" />
  <!-- remap topics to conform to ROS-I specifications -->
  <rosparam file="$(find abb_irb120_gazebo)/config/joint_state_controller.yaml"
command="load" />
  <node name="joint_state_controller_spawner" pkg="controller_manager"
type="controller_manager" args="spawn joint_state_controller" />
```

Figura 5.22: Recorte del archivo "irb120_3_58_gazebo.launch".

Se ha aprendido a utilizar la herramienta “group", que se utiliza si se quiere duplicar la información, pero agrupándola en variables con diferentes variables, como se aprecia en la Figura 5.22. En este grupo se carga el archivo URDF, se describe una posición inicial para evitar colisiones entre ambos robots y se cargan los controladores. Se crean dos grupos, uno para cada robot. El segundo grupo tendrá la misma estructura salvo que se modifica su posición inicial y se cambian los nombres de los controladores y del modelo para evitar confusiones. (Se adjudica un número aleatorio, al final de cada nombre, como por ejemplo “abb_irb_3_58_1”).

Al tener el programa sin errores, se lanza, tal y como se puede observar en la Figura 5.23 se han duplicado todos los parámetros. También se puede observar que los nodos también han sido duplicados y cada uno posee un nombre diferente.

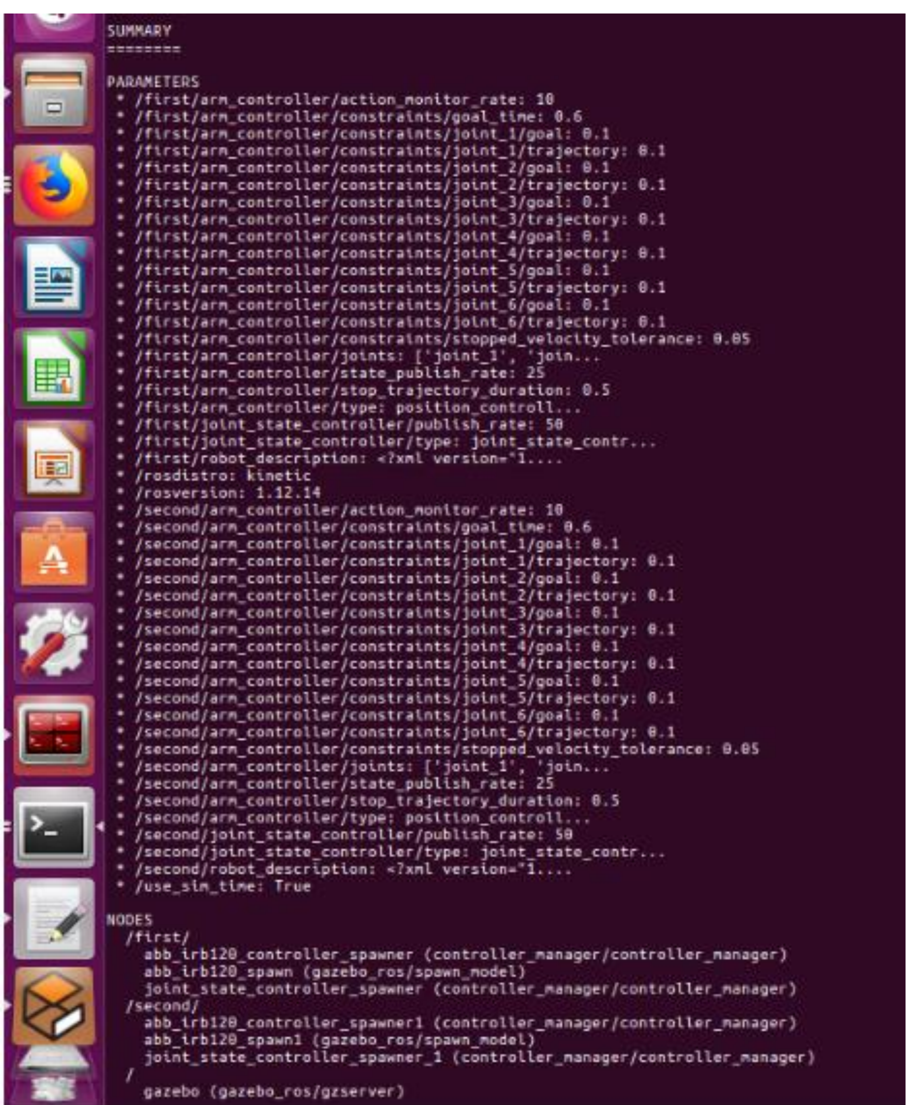


Figura 5.23: Parámetros de cada robot al lanzar el archivo “irb120_3_58_gazebo.launch”.

Para poder mover los robots basta con crear una llamada al "topic" correspondiente a cada robot. Se utiliza el mismo comando de la Figura 5.16 pero con una diferencia. Al crear la variable para llamar al "topic", se escribirá uno de los dos "topics" siguientes en función del robot que se desee mover:
"rospy.Publisher('/first/arm_controller/command')"
"rospy.Publisher('/second/arm_controller/command')"

Si se quiere mover los dos robots con el mismo código se crean dos variables diferentes ("pub_1", y "pub" por ejemplo) para poder enviar dos trayectorias distintas ya que si no se moverían ambos robots al mismo punto.

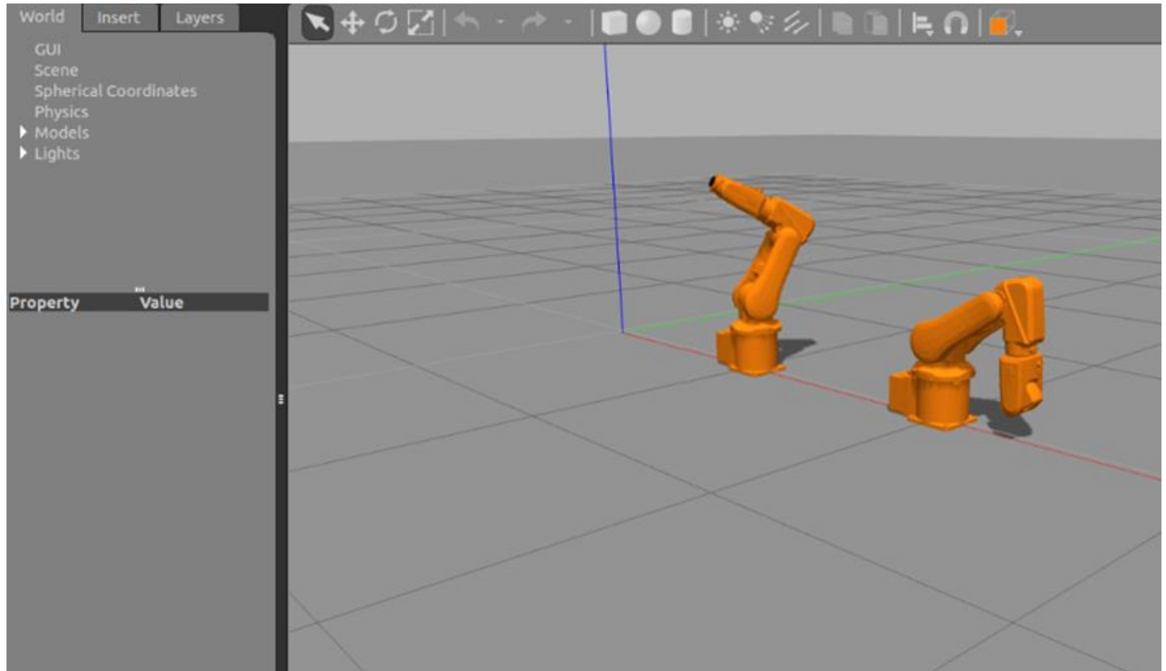


Figura 5.24: Simulación en Gazebo de dos robots tras haber ejecutado dos trayectorias diferentes.

Tras la lectura de este Capítulo el lector es capaz de ejecutar los movimientos que desee en un robot cualesquiera, pero sin poder realizar ninguna tarea funcional ya que los robots no contienen las herramientas necesarias. Se ha dado un paso más y se han aplicado los métodos en una simulación multirobot, lo que proporciona una gran versatilidad si lo que se quiere es trabajar con una célula de trabajo compuesta de varios robots.

6. Tareas de manipulación con robot colaborativo

En este sexto capítulo, se describe cómo realizar dos tareas de manipulación con un robot colaborativo. Primeramente, se hace una pequeña introducción sobre la empresa que fabrica estos robots. Las dos manipulaciones se realizan con un modelo UR5, pero tienen diferencias entre ellas. En la primera, se utiliza una pinza robótica para agarrar el objeto, mientras que, en la segunda, la herramienta es un "vacuum gripper". En ambas simulaciones, se han implementado distintas aplicaciones para conseguir transportar un objeto o pieza de un lugar a otro.

6.1 UNIVERSAL ROBOTS

Un cobot o robot colaborativo es un robot creado para interactuar con los humanos en un entorno de trabajo. El término se utilizó por primera vez en 1999 y surge de la unión de las palabras "colaboración" y "robot". Sus dos principales características son:

- **Colaboración:** Los cobots fueron creados para ayudar a las personas, por lo tanto, la interacción con los operarios es fundamental.
- **Automatización:** Un robot colaborativo es capaz de realizar tareas de automatización inimaginables para un robot industrial, logrando impulsar la productividad a otros niveles.

Una de las empresas con más renombre en el sector de la robótica colaborativa es Universal Robots, gracias a la fabricación y puesta en marcha de sus tres modelos de cobots. Las diferencias entre modelos se pueden observar en la Figura 6.1.

El modelo UR3 es más pequeño de la gama y es una buena opción para tareas y trabajos ligeros de ensamblado, o trabajos que requieren una precisión absoluta. El UR5 es ligeramente más grande y perfecto para automatizar las tareas de procesamiento de poco peso, como "pick & place", ensayos, y pruebas en laboratorios. Por último, el modelo UR10 además de ser el brazo robótico de mayor tamaño en la familia UR y el más potente, ofrece un elevado grado de precisión. Este brazo robótico colaborativo permite automatizar procesos y tareas con un peso de hasta 10 kg [16].

Huella	128 mm.	149 mm.	190 mm.
Alcance	500 mm.	850 mm.	1300 mm.
Carga útil	3 kg.	5 kg.	10 kg.
Peso	11 kg.	18,4 kg.	33,5 kg.

Figura 6.1: Proporción de tamaños entre cada uno de los tres modelos de Universal Robots con sus respectivas características [16].

Universal Robots es una empresa danesa, que fabricó su primer cobot con forma de brazo robótico articulado hace diez años. Ha ido ganando importancia en el sector de la robótica colaborativa, sobre todo por las características que poseen los tres modelos que fabrican, algunas de ellas son las siguientes:

- **Fáciles de programar:** A cada cobot va conectada una "Tablet" donde se puede programar mediante una intuitiva interfaz con visualización tridimensional.
- **Instalación rápida:** El tiempo de instalación medio es de tan solo medio día.
- **Flexibles:** Los tres modelos se adaptan fácilmente a cualquier entorno ya que pueden cambian de proceso de forma rápida y sencilla.
- **Colaborativos y seguros:** Pueden compartir espacio de trabajo junto a operarios sin necesidad de estar vallados y son más seguros que los robots industriales de mayor tamaño. Su sistema de seguridad está aprobado y certificado por la asociación alemana de inspección técnica llamada "TÜV".

- **Aplicaciones de todo tipo:** Los cobots de Universal Robots son capaces de desempeñar una gran cantidad de aplicaciones de todo tipo. Sobre todo, debido a la facilidad por la que se puede dotar de sensores a estos robots (cámaras, “grippers” etc.).

La programación de cualquiera de los tres modelos se suele realizar a través de una “Tablet” como la de la Figura 6.2 que va conectada a cada cobot. No es el único método ya que como se verá a continuación, mediante ROS también se podrá programar y simular un robot colaborativo de Universal Robots.

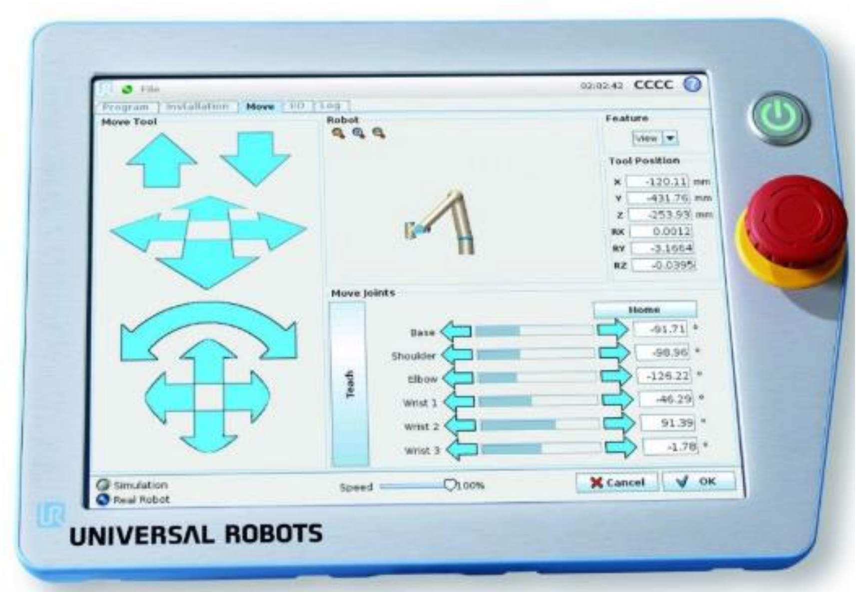


Figura 6.2: “Tablet” usada para programar cobots de Universal Robots [16].

6.2 MANIPULACIÓN CON PINZA ROBÓTICA

En este segundo apartado, se describe como realizar una tarea de manipulación de un objeto, mediante un robot colaborativo y una pinza robótica. Se ha optado por trabajar con el modelo UR5 (se describen sus principales características técnicas en el Anexo B), el cual, posee un valor añadido, ya que en su extremo (“Joint_6”) se le ha programado una pinza robótica 2F-85 de la marca “Robotiq”. Para llevar a cabo su manipulación, lo primero es cargar los paquetes que contienen los archivos necesarios para la simulación del robot colaborativo funcional.

Se ejecutan los siguientes comandos para copiar los dos paquetes en el directorio

principal, y posteriormente compilarlos.

```
$ cs (comando rápido para acceder a la carpeta "/catkin_ws/src")
$ git clone https://github.com/utecrobotics/ur5
$ git clone https://github.com/utecrobotics/robotiq
$ cd ~/catkin_ws && catkin_make
```

En la Figura 6.3 se muestra la simulación que está implementada en el código compilado, es decir, el robot colaborativo (modelo UR5), su respectiva herramienta (Pinza robótica 2F-85) y la mesa sobre la que se sitúa el robot. Para la simulación es necesario lanzar el siguiente comando.

```
$ roslaunch ur5_gazebo ur5_setup.launch
```

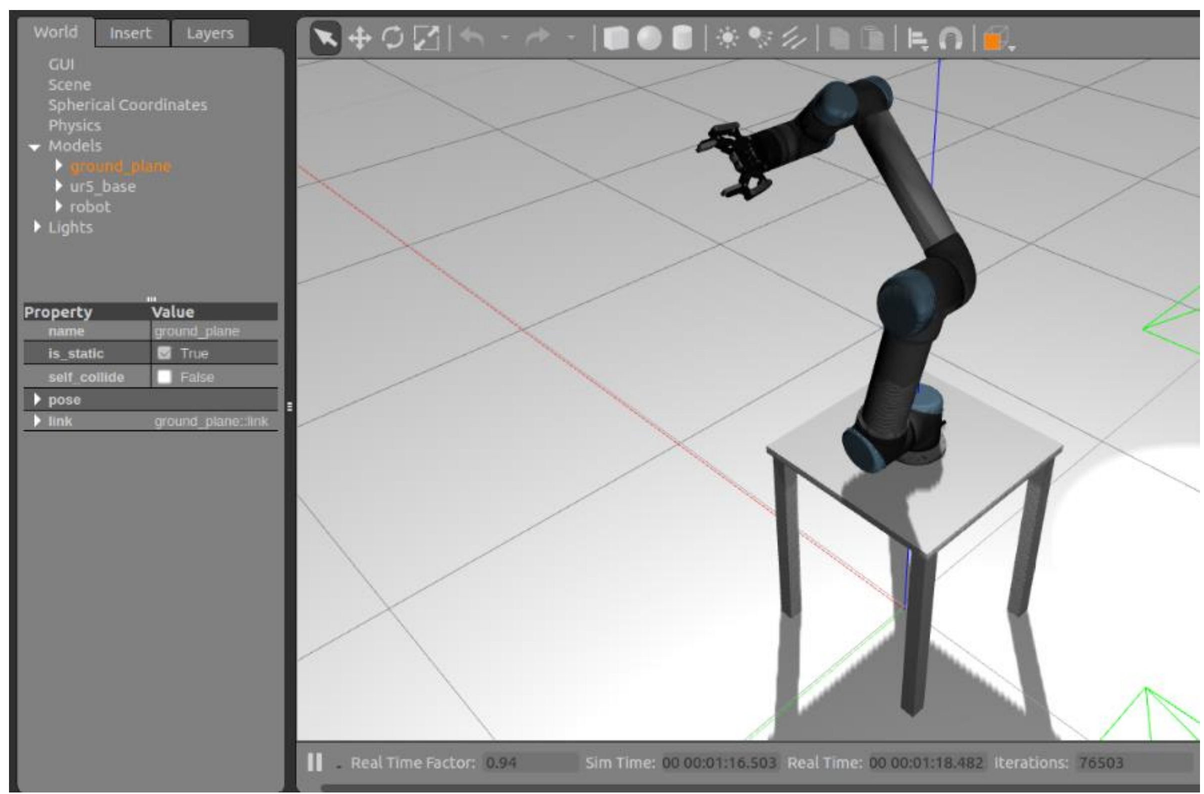


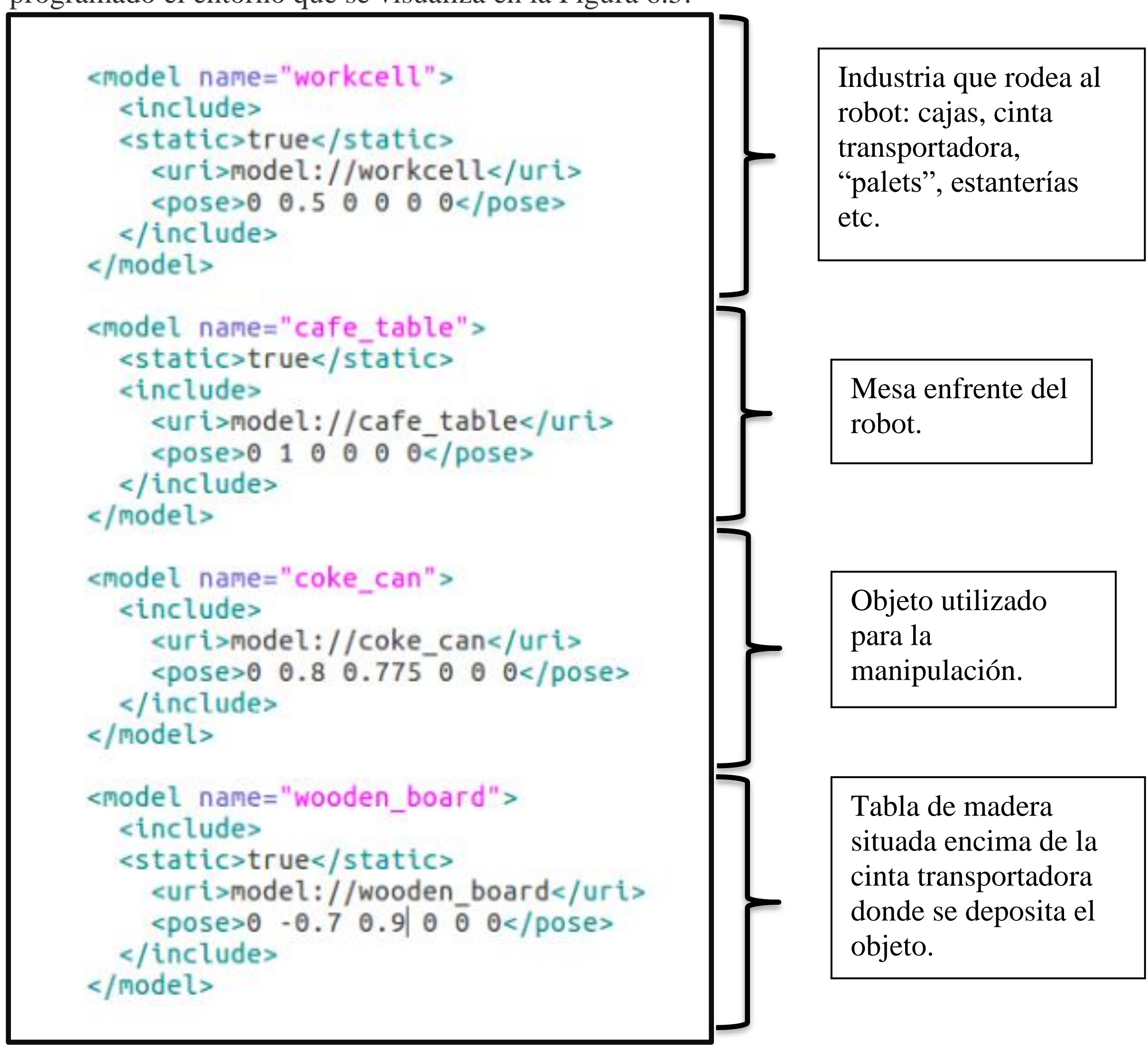
Figura 6.3: Simulación en Gazebo del modelo UR5 y la pinza robótica 2F-85.

El siguiente paso es crear un marco o ambiente en el cual se realizará la tarea de manipulación. Para ello, Gazebo cuenta con un gran número de modelos ya creados que se pueden simular introduciendo las líneas de programa de la Figura 6.4 en el archivo "ur5_setup.world". Si no fuera suficiente con éstos, se pueden crear fácilmente nuevos modelos o cargar otros creados por usuarios de esta plataforma. En este caso, el primero de los modelos ("workcell"), creado para la competición ARIAC (se amplía la

información en el Capítulo 6.3), para su simulación se tiene que crear una carpeta y copiar el código del modelo en ella. El resto de los modelos son cargados de los predeterminados de Gazebo.

```
$ mkdir -p ~/.gazebo/models
```

El anterior comando se utiliza para crear la carpeta que contendrá los archivos de los modelos que se quieran simular, tanto los que cree uno mismo como los creados por otras personas. En la Figura 6.4 se muestran estos modelos en un archivo, que contiene programado el entorno que se visualiza en la Figura 6.5.



```
<model name="workcell">
  <include>
  <static>true</static>
    <uri>model://workcell</uri>
    <pose>0 0.5 0 0 0 0</pose>
  </include>
</model>

<model name="cafe_table">
  <static>true</static>
  <include>
    <uri>model://cafe_table</uri>
    <pose>0 1 0 0 0 0</pose>
  </include>
</model>

<model name="coke_can">
  <include>
    <uri>model://coke_can</uri>
    <pose>0 0.8 0.775 0 0 0</pose>
  </include>
</model>

<model name="wooden_board">
  <include>
  <static>true</static>
    <uri>model://wooden_board</uri>
    <pose>0 -0.7 0.9 0 0 0</pose>
  </include>
</model>
```


Figura 6.4: Recorte del archivo “ur5_setup.world”.

Se han ido ajustando las posiciones mediante diferentes pruebas y a todos los modelos, salvo la lata, se les aplica una condición para ser estáticos y que no se muevan. (condición aplicada con “<static>true</static>”)

Tras haber creado el entorno, se procede a programar la acción que se desea realizar con el robot. En el caso que nos ocupa, la definición de la tarea es el transportar una lata de refresco, hasta situarla sobre una tabla, localizada encima de la cinta transportadora. Para ello la acción consta de cinco partes.

- Primeramente, el robot se acerca al objeto, quedando la lata situada entre la pinza para poder agarrarla.
- El “gripper” se cierra y atrapa a la lata.

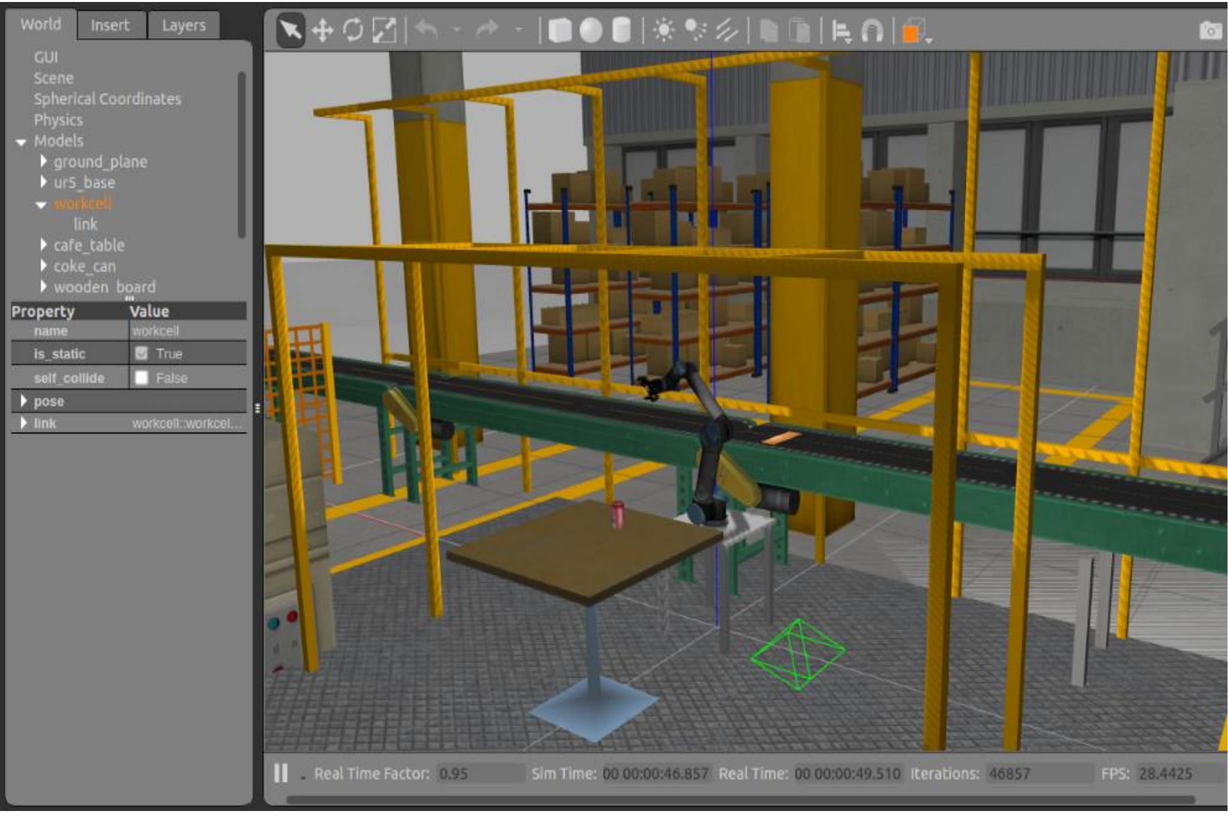


Figura 6.5: Simulación al crear el entorno y antes de ejecutar la tarea

- El robot se mueve con la lata atrapada entre la pinza, hasta situar la lata sobre la tabla de madera.

- El “gripper” se abre.

- Con dos movimientos el robot vuelve a la posición de origen. En el primero, la pinza se levanta a una posición segura donde no pueda tirar la lata en su movimiento de retorno. En el segundo se mueve libremente

hasta la posición de partida.

Para la ejecución de la tarea es suficiente con ejecutar los siguientes comandos cada uno en una terminal diferente:

```
$ roslaunch ur5_gazebo ur5_setup.launch
$ rosrun ur5_gazebo tarea.py
```

El primero de los comandos se encarga se cargar el robot en el entorno programado, mientras que el segundo ejecuta el archivo Python donde está descrita la tarea. Este archivo viene completo y descrito con detalle en el Anexo D. La Figura 6.6 muestra una captura de la tercera parte de la tarea, cuando el robot traslada la lata.

Figura 6.6: Imagen del robot durante su acción de llevar la lata sobre la tabla.

6.3 MANIPULACIÓN CON "VACUUM GRIPPER"

Para llevar a cabo la manipulación con un "vacuum gripper" se ha utilizado un modelo de "ARIAC" (Agile Robotics for Industrial Automation Competition). ARIAC es una competición anual organizada por NIST (National Institute of Standards and Technology) [18], un instituto tecnológico de Estados Unidos. La competición, dotada

de una recompensa económica, consiste en programar una tarea de manipulación con ROS y sus herramientas. La meta del concurso es probar la agilidad de los sistemas de robots manipuladores, con el objetivo de permitir que los robots en las industrias sean más productivos, más autónomos y respondan mejor a las necesidades de los trabajadores. La simulación del ganador está en código libre en la comunidad de ROS, para que los usuarios puedan usar dicho entorno. En este caso, se utiliza el modelo de ARIAC 2017 [17], y se escribe un archivo en Python para realizar una tarea de manipulación.

Se comienza por instalar los paquetes de ARIAC 2017, en un nuevo directorio llamado “ariac_ws”.

$ mkdir -p ~/ariac_ws/src (se crea el directorio “ariac_ws” y la carpeta “src”)

$ cd ~/ariac_ws/src (se accede a la carpeta “src”)

$ git clone https://bitbucket.org/osrf/ariac -b ariac_2017 (se copia el paquete ARIAC 2017)

A diferencia de los paquetes vistos anteriormente, en este caso no se compila, sino que se instala mediante el siguiente comando:

$ cd ~/ariac_ws/src (se accede a la carpeta “src”)

$ catkin_make install

Cuando el paquete se instale sin errores se generan tres carpetas, dos de ellas se han visto previamente (“build” y “devel”), y una es nueva (“install”). La nueva carpeta contiene todos los archivos ejecutables. Por lo tanto, si se quiere modificar algún parámetro de la simulación se tiene que acudir a esta carpeta. En caso de crear un nuevo archivo, este también deberá ser copiado dentro de la carpeta “install”.

ROS no permite trabajar con dos espacios de trabajo a la vez, sino que es necesario ir alternando cada vez que se utilice uno de ellos. Como en este apartado solo se utiliza el espacio “ariac_ws”, se configura el archivo “.bashrc” para trabajar solo con dicho directorio. Se abre el archivo con un editor de texto *($ gedit .bashrc)* y se escribe el siguiente código.

$ source ~/ariac_ws/install/setup.bash

Si mientras se utiliza el espacio de trabajo "ariac_ws", se quiere trabajar en algún momento puntual con el directorio "catkin_ws", basta con escribir este comando en una nueva terminal (Cuando se cierra esta terminal, automáticamente se vuelve a trabajar en el espacio "ariac_ws").

```
$ source ~/catkin_ws/devel/setup.bash
```

La simulación se lanza mediante este código [19]:

```
$ rosrun osrf_gear gear.py --development-mode -f `catkin_find --share osrf_gear`/config/escenario_1.yaml
```

El archivo "gear.py" contiene los algori para ejecutar la simulación en Gazebo, mientras que con el archivo "escenario_1.yaml" se configuran los sensores, robot y herramientas. Se han modificado varios sensores y herramientas con respecto al archivo original llamado "sample.yaml".

```
models_over_bins:
  bin7:
    models:
      piston_rod_part:
        xyz_start: [0.1, 0.1, 0.0]
        xyz_end: [0.5, 0.5, 0.0]
        rpy: [0, 0, 'pi/4']
        num_models_x: 3
        num_models_y: 4
  bin8:
    models:
      piston_rod_part:
        xyz_start: [0.1, 0.1, 0.0]
        xyz_end: [0.5, 0.5, 0.0]
        rpy: [0, 0, 'pi/4']
        num_models_x: 3
        num_models_y: 4
  bin6:
    models:
       gear_part:
        xyz_start: [0.1, 0.1, 0.0]
        xyz_end: [0.5, 0.5, 0.0]
        rpy: [0, 0, 0]
        num_models_x: 4
        num_models_y: 4
sensors:
  break_beam:
    type: break_beam
    pose:
      xyz: [1.6, 2.25, 0.91]
      rpy: [0, 0, 'pi']
  proximity_sensor:
    type: proximity_sensor
```

Nuevas piezas situadas en las cajas 8 y 6.

Figura 6.7: Fragmento del archivo "escenario_1.yaml"

Tal y como se muestra en la Figura 6.7, se han introducido piezas en las cajas 6 y 8, mediante un sistema de matriz 3x4 y 4x4, con respecto al modelo original. También se han eliminado varios sensores ya que no se utilizan para la tarea que se va a realizar. El resultado de estas modificaciones se visualiza en la Figura 6.8.

Figura 6.8: Escenario previo a la realización de la tarea de manipulación.

La tarea consiste en depositar una pieza de la caja Nº7 (la más cercana al robot) en el AGV Nº1 (situado a la izquierda de la simulación). Una vez depositada la pieza, el AGV ("Automated guided vehicle") comienza su movimiento para descargar la pieza y situarse en su posición de partida totalmente vacío. El robot mientras, se acerca a la cinta transportadora para llevar una de las piezas que avanzan por la cinta al AGV Nº2. El robot vuelve a la posición de inicio mientras el AGV Nº2 descarga la pieza depositada. La tarea se puede dividir en seis partes:

-El robot se acerca a la caja Nº7.

-Se activa el "vacuum gripper" que, mediante aire, succiona la pieza para que el robot la desplace.

-El robot se mueve hasta el AGV Nº1

-Se desactiva el "vacuum gripper" y la pieza queda depositada sobre el AGV

-El AGV Nº1 descarga la pieza y retorna a la posición de partida vacío mientras el robot transporta una de las piezas de la cinta al AGV Nº2.

-El robot vuelve a la posición de inicio mientras el AGV Nº2 descarga la pieza.

La Figura 6.9 muestra como el robot está desplazando la pieza hasta el AGV Nº1. Posteriormente la depositará en el AGV.

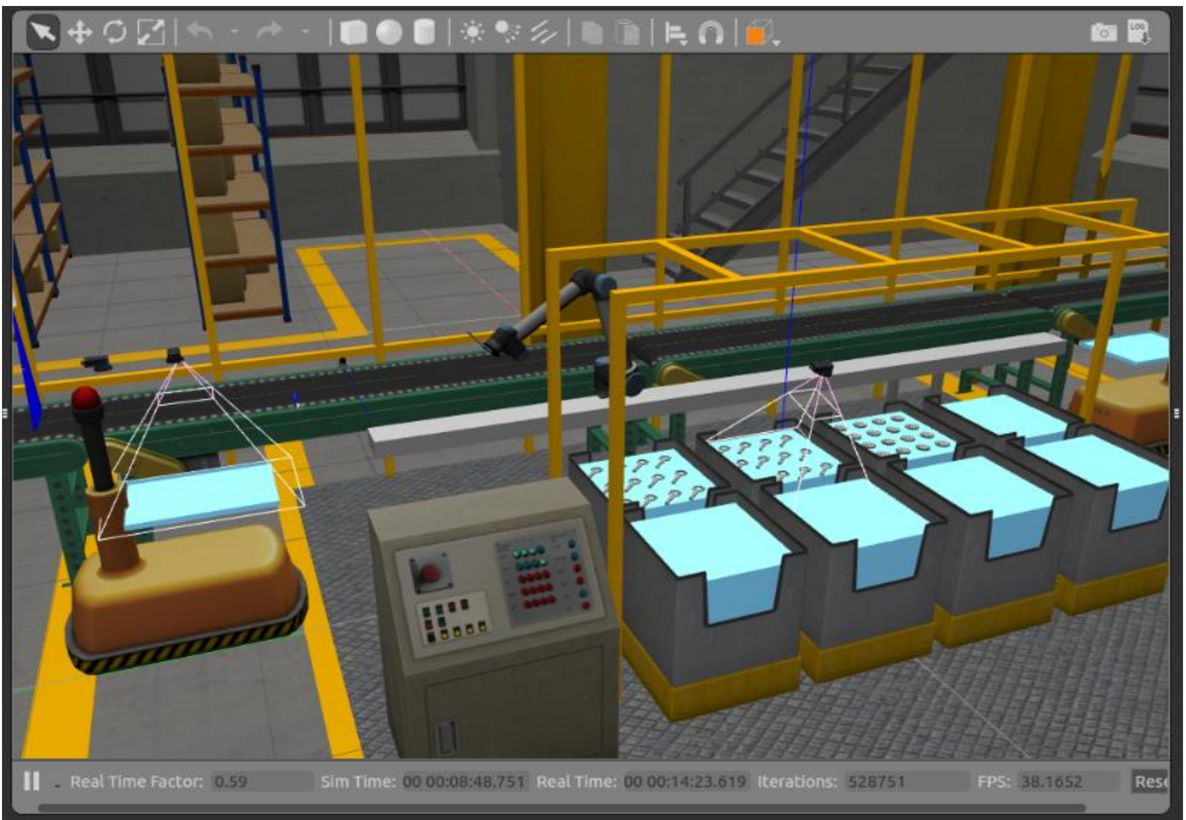


Figura 6.9: Robot desplazándose hasta el AGV Nº1.

Para ejecutar la tarea se escriben los siguientes comandos en diferentes terminales:

```
$ rosrun ariac_example ariac_example_node
```

```
$ rosrun tarea_1.py
```

El primer comando es un programa que viene ya implementado en el paquete de ARIAC, se encarga de devolver la posición de los “joints” del robot para así poder ver como va modificando su posición. También se encarga de activar la cinta transportadora.

El segundo comando ejecuta el archivo que se ha implementado a mano en Python, que viene recogido y explicado en el Anexo E.

7. Conclusiones y líneas futuras.

En este último capítulo de la memoria, se presentan algunas conclusiones sobre el trabajo realizado a lo largo de este proyecto. Además, se proponen una serie de posibles mejoras que se podrían llevar a cabo en el futuro.

7.1 CONCLUSIONES

Las **conclusiones** que se desprenden del presente Trabajo Fin de Grado abarcan diferentes ámbitos:

La conclusión más importante, es que se ha cumplido con creces el principal objetivo marcado para este Trabajo Fin de Grado. Se ha conseguido realizar varias tareas de manipulación con ROS, en un entorno de simulación, con diferentes robots industriales.

Para llegar a este objetivo final se han ido cumpliendo satisfactoriamente todos los pasos, comenzando por una correcta instalación y configuración de los programas utilizados. El primer paso ha sido conseguir mover un robot IRB120 de ABB en un entorno de simulación mediante diferentes métodos. Es interesante mencionar el método con "MoveIt" ya que, tras haber configurado el robot a nuestro gusto, se ha comprobado que estas configuraciones se ven aplicadas a la hora de ejecutar una simulación del robot. Se ha sido capaz de trasladar movimientos de una herramienta a otra, es decir, de ejecutar un movimiento en RViz y visualizarlo en Gazebo.

Pero es más importante destacar el método en el cual, partiendo de cero, se escribe un programa funcional en Python para enviarle la trayectoria deseada al robot, ya que es el utilizado en los demás apartados del Trabajo. Posteriormente, modificando este algoritmo se consigue programar los movimientos de varios robots en un mismo entorno. Estos movimientos tienen la función de entender las diferentes formas que se pueden utilizar para programar movimientos y cual será la más apropiada a la hora de realizar una aplicación.

A continuación, se cumple uno de los objetivos más complejos, que es el de programar una tarea de manipulación con el modelo UR5 de Universal Robots. Para ello, es necesario incluir todas las trayectorias que el robot tiene que realizar y ejecutar en un mismo archivo una función para la herramienta. En este primer caso, como la herramienta es una pinza, la función se encarga de enviarle el valor de la apertura. El programa descrito viene recogido en el Anexo D. También se comprueba como con el código utilizado en los anteriores apartados se puede aplicar para otro modelo de robot.

Por último, se programa otra tarea de manipulación con diferentes funciones. Al ser la herramienta del UR5 un “vacuum gripper” una de las funciones será para activarlo o desactivarlo. Otra de las funciones activará el movimiento de dos AGVs que vienen programados por defecto en el entorno de simulación. Este programa viene descrito en el Anexo E.

7.2 LÍNEAS FUTURAS

Las **líneas futuras** en las que se pueden embarcarse nuevas investigaciones acerca de los temas tratados en este trabajo son varias.

La primera es continuar con la idea de la simulación multirobot. Cabría la posibilidad de que varios robots realizaran diferentes tareas de manipulación en un mismo entorno. A partir del trabajo realizado en este proyecto, sería necesario la programación de otro robot y con el mismo código modificar varios parámetros (“topics”, posición de inicio, vectores de posición, etc.). Esto sería de gran utilidad ya que como se explica anteriormente los robots industriales suelen trabajar en células de trabajo realizando varias acciones simultaneas o en serie.

Otra opción de investigación seria mejorar la programación en Python. Por ejemplo, utilizar la cinemática inversa para enviarle los movimientos a los robots. Esta cinemática tiene el problema de que es compleja de programar, pero una vez programada se obtienen mejores resultados (aumento de la precisión, mayor rapidez de movimientos etc).

Utilizar los sensores que vienen implementados en el entorno de ARIAC sería de gran utilidad a la hora de realizar una aplicación más compleja. Por ejemplo, en la Figura 6.8 se aprecian varias cámaras encima de los AGVs o de las cajas. También hay sensores de movimientos a lo largo de la cinta transportadora que se podrían utilizar para avisar al robot de ejecutar una acción.

Finalmente, y hablando un poco más personal, me gustaría participar en la competición de ARIAC. Con lo aprendido en este Trabajo estoy capacitado para programar una aplicación robótica en una simulación. Por lo tanto, ¿por qué no presentarme e intentar ganar el primer premio de 17.500$ que ofrece el National Institute of Standards and Technology a la mejor aplicación?

BIBLIOGRAFÍA

[1] "Clasificación de robots" *https://www.clasificacionde.org/*

[2] "Robots según su función" *https://www.esneca.com/blog/clasificacion-de-los-robots-segun-su-funcion/*

[3] "Robots industriales más utilizados" *https://www.bfmx.com/tipos-de-robots-industriales-mas-utilizados/*

[4] "Aplicaciones de robots industriales" *http://platea.pntic.mec.es/vgonzale/cyr_0204/cyr_01/robotica/aplicaciones.htm*

[5] "Similitudes entre el brazo humano y el brazo robótico" *http://electronica-jaimes.blogspot.com/*

[6] "Robots colaborativos" *https://www.asepeyo.es/blog/robots-colaborativos/*

[7] García Cazorla, Alvaro. (2013) *ROS: Robot Operating System* (Trabajo Fin de Grado) Universidad politécnica de Cartagena, Murcia.

[8] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung and TaeHoon Lim, ROS, Robot Programming,", pp. 1-308, 2017.

[9] Gómez Rubio, Alejandro. (2014) *Control de la mano robótica BarrettHand BH8-262 en entorno ROS* (Trabajo Fin de Grado) Universidad de Alcalá Escuela Politécnica Superior, Madrid.

[10] Lentin Joseph, *Mastering ROS for Robot Programming, Design, build, and simulate complex robots using Robot Operating System and master its out-of-the-box functionalities* ", pp. 1-155, 2015.

[11] "Gazebo, simulador multi-robot 3D." *http://gazebosim.org/.*

[12] "ROS Visualization" *https://wiki.ros.org/rviz.*

[13] Buenavida Durán, Francisco Javier. (2017) *Programación de robot móvil con manipulador para el sector del comercio en entorno ROS* (Trabajo Fin de Máster) Escuela Técnica Superior de Ingeniería, Sevilla.

[14] "MoveIt" *https://moveit.ros.org*

[15] "Documentación ABB" *https://new.abb.com/es*

[16] "Documentación Universal Robots" *https://www.universal-robots.com/es/*

[17] "ARIAC 2017" *https://bitbucket.org/osrf/ariac/wiki/2017/Home*

[18] "National Institute of Standards and Technology" *https://www.nist.gov/*

[19] "Tutoriales ARIAC" *http://wiki.ros.org/ariac/Tutorials/*

ANEXOS

A. CARACTERÍSTICAS TÉCNICAS DEL ROBOT INDUSTRIAL IRB120

A continuación, se describen algunas de las características más relevantes del robot industrial IRB120, comenzando por visualizar sus dimensiones en la Figura A.1

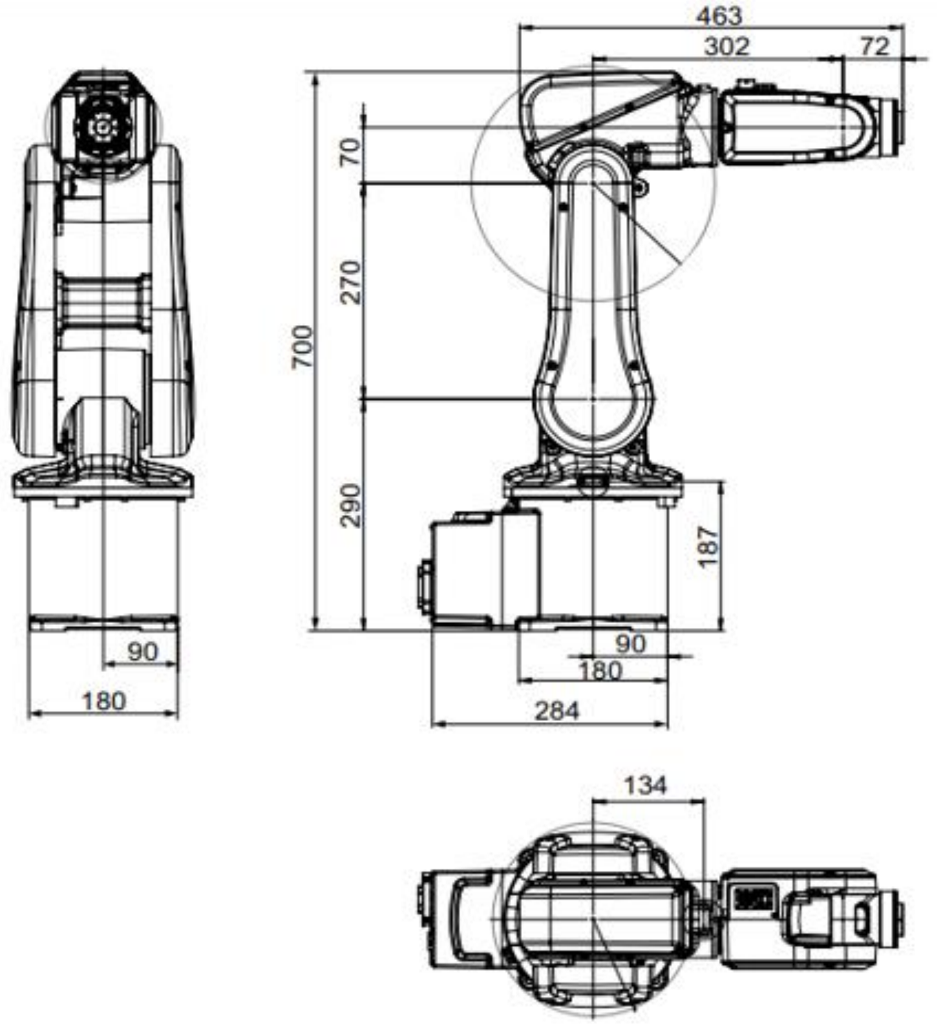


Figura A.1: Dimensiones en milímetros del robot IRB120 [15].

Este robot multiusos fabricado por ABB es el de menor tamaño y peso ya que tan sólo pesa 25 kg. Es capaz de soportar una carga de 3 kg (4 kg en posición vertical de la muñeca) con un alcance de 580 mm.

El controlador que utiliza es el “IRC5” (Single Cabinet Controller) que contiene todas las funciones necesarias para mover y controlar el robot. Este controlador compacto se utiliza también con varios modelos de ABB como son el IRB1520, IRB1600 y IRB360.

El robot está equipado con el software de control de robots RobotWare, que admite todos los aspectos del sistema de robot, como el control del movimiento, el desarrollo y la ejecución de programas, la comunicación, etc.

Las velocidades varían en función de los ejes. Para los tres primeros ejes la velocidad es de 250º/s, mientras que para el cuarto y el quinto es de 320º/s. La mayor velocidad se alcanza en el sexto y último eje ya que es de 420º/s. Se puede apreciar dónde está situado cada eje en la Figura A.2.

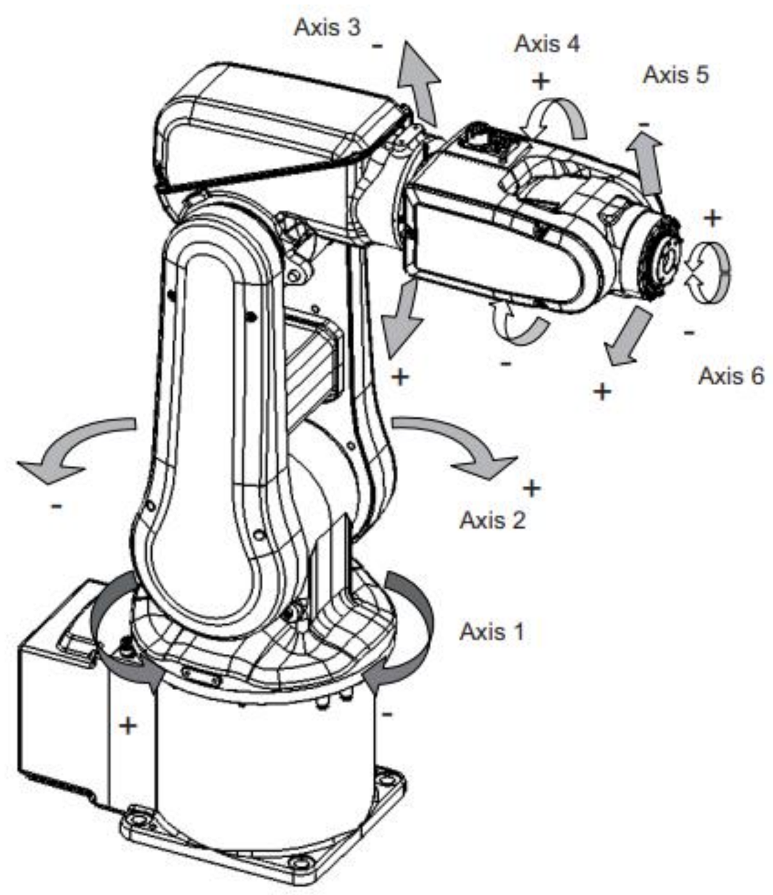


Figura A.2: Boceto del robot IRB120 con la diferenciación entre ejes [15].

B. CARACTERÍSTICAS TÉCNICAS DEL ROBOT COLABORATIVO UR5

En este segundo Anexo se describen los datos técnicos más característicos del modelo UR5, fabricado por Universal Robots.

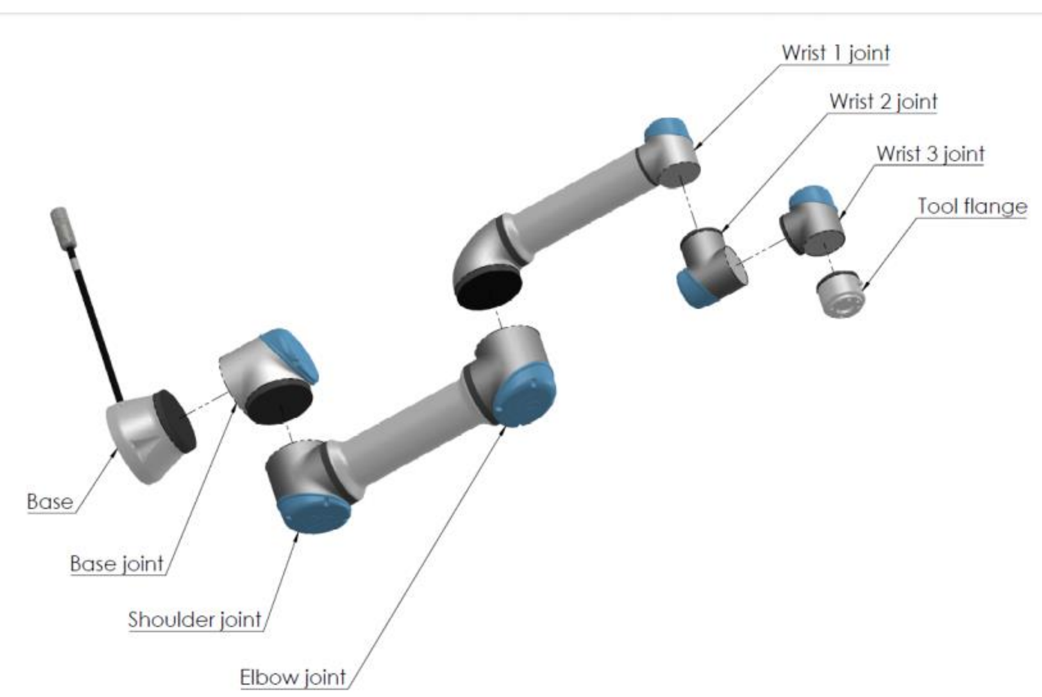


Figura B.1: Despiece del modelo UR5 [16].

Este modelo “intermedio” de la empresa Universal Robots tiene un peso de 18,4 kg., y es capaz de soportar una carga de 5 kg. Como se aprecia en la Figura B.1, posee 6 juntas giratorias (“Base Joint”, “Shoulder Joint”, “Elbow Joint”, “Wrist 1 Joint”, “Wrist 2 Joint” y “Wrist 3 Joint”) con una repetibilidad de ± 0.1 mm. El radio de giro de cada articulación es de 360º con una velocidad máxima de 180º/segundo. Es importante también, su reducido consumo de energía con respecto a otros robots de similares características. En este caso, el consumo es de 200 W, utilizando un programa típico.

En el desarrollo de este brazo robótico se aplican numerosas normas europeas (EN), como internaciones (ISO). La conformidad con estas normas garantiza que se siguen todas las instrucciones de montaje, instrucciones de seguridad y directrices del manual del usuario del modelo UR5. Se destaca una, que regula todas las operaciones de colaboración y que es la ISO 10218-1:2011.

ANEXO C

C. SOLUCIÓN DE ERRORES

No todo ha ido sobre ruedas en este proyecto. Durante su elaboración han ido surgiendo numerosos problemas y dificultades ya sea a la hora de instalar programas, compilar códigos o cargar librerías. Debido a ese motivo se realiza un Anexo donde se recogen parte de los errores que se han ido sucediendo a lo largo del trabajo con sus respectivas soluciones.

- **ERROR DE INSTALACION DE ROS:**

El primer error solucionado aparece nada más comenzar el proyecto, durante la instalación de ROS al ejecutar el comando para instalar la versión completa.

```
$ sudo apt-get install ros-kinetic-desktop-full
Reading package lists... Done
Building dependency tree
Reading state information... Done
Some packages could not be installed. This may mean that you have requested an impossible situation or if you are using the unstable distribution that some required packages have not yet been created or been moved out of Incoming.
The following information may help to resolve the situation:

The following packages have unmet dependencies:
 ros-kinetic-desktop-full: Depends: ros-kinetic-desktop but it is not going to be installed
                           Depends: ros-kinetic-perception but it is not going to be installed
                           Depends: ros-kinetic-simulators but it is not going to be installed
                           Depends: ros-kinetic-urdf-tutorial but it is not going to be installed
E: Unable to correct problems, you have held broken packages
```

Para solucionarlo se tiene que modificar el archivo “Sources.list”, donde se recogen los repositorios o fuentes de los paquetes de software que pueden ser actualizados, instalados, eliminados etc. Se ejecuta el siguiente comando para editar el archivo.

```
$ sudo gedit /etc/apt/sources.list
```

ANEXO C

Una vez abierto el programa se copian las siguientes líneas:

-deb http://cz.archive.ubuntu.com/ubuntu xenial main universe
-deb http://mirrors.ustc.edu.cn/ubuntu/ xenial main restricted universe multiverse
-deb http://mirrors.ustc.edu.cn/ubuntu/ xenial-security main restricted universe multiverse
-deb http://mirrors.ustc.edu.cn/ubuntu/ xenial-updates main restricted universe multiverse
-deb http://mirrors.ustc.edu.cn/ubuntu/ xenial-proposed main restricted universe multiverse
-deb http://mirrors.ustc.edu.cn/ubuntu/ xenial-backports main restricted universe multiverse
-deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial main restricted universe multiverse
-deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial-security main restricted universe multiverse
-deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial-updates main restricted universe multiverse
-deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial-proposed main restricted universe multiverse
-deb-src http://mirrors.ustc.edu.cn/ubuntu/ xenial-backports main restricted universe multiverse

Son archivos que contienen paquetes binarios (deb), esto es, los paquetes pre-compilados que normalmente se usan, o contienen los paquetes fuente (deb-src). Cuando se guarda el archivo y se cierra se ejecuta este comando para actualizar los paquetes, seguidamente se vuelve a lanzar el comando para la instalación de ROS, que funcionará sin problemas.

```
$ sudo apt-get update
$ sudo apt-get install ros-kinetic-desktop-full
```

- **ERROR CON LOS PAQUETES DEL “TURTLEBOT3”:**

El siguiente error surge al compilar los paquetes necesarios para poder simular el “Turtlebot3”. Tras copiar los cuatro paquetes (Figura 4.5) se ejecuta el comando para compilarlo y aparece el siguiente error.

```
$ cd ~/catkin_ws && catkin_make
CMake Error at /opt/ros/kinetic/share/catkin/cmake/catkinConfig.cmake:83 (find_package):
  Could not find a package configuration file provided by
  "gazebo_ros_control" with any of the following names:

    gazebo_ros_controlConfig.cmake
    gazebo_ros_control-config.cmake
```

Esto quiere decir que para poder compilarlos sin problemas es necesario copiar en el directorio otro paquete que a priori no era necesario. Se copia el paquete solicitado y se relanza el código.

```
$ cd ~/catkin_ws/src/
$ git clone https://github.com/ros-simulation/gazebo_ros_pkgs.git
$ catkin_make
```

- **ERROR AL INTENTAR INSTALAR APLICACIONES**

A pesar de que este error no es de ROS se incluye ya que se sucede varias veces al instalar aplicaciones como Gazebo o MoveIt. Al lanzar el siguiente comando para instalar un programa aparece este error:

```
$ sudo apt-get [aplicación que se quiere instalar]
E: Could not get lock /var/lib/dpkg/lock - open (11: Resource temporarily unavailable)
E: Unable to lock the administration directory (/var/lib/dpkg/), is another process using it?
```

Puede haber varias razones por las que sucede este error. La primera de ellas podría ser que algún otro proceso está usando la herramienta de administración de paquetes APT (apt o apt-get en otras palabras) y basta con matar todas las instancias del programa y volverlo a lanzar.

```
$ sudo killall apt apt-get
$ sudo apt-get [aplicación que se quiere instalar]
```

La segunda razón es que se tenga un archivo de bloqueo activo. Estos archivos se utilizan para evitar que dos o más procesos utilicen los mismos datos. Cuando apt se ejecuta, crea archivos de bloqueo en algunos lugares. Cuando el comando apt anterior no finaliza correctamente, los archivos de bloqueo no se eliminan y, por lo tanto, impiden nuevas instancias de los comandos apt / apt-get.

Para solucionar el problema, se tienen que eliminar los archivos de bloqueo utilizando los siguientes comandos:

```
$ sudo rm /var/lib/apt/lists/lock
$ sudo rm /var/cache/apt/archives/lock
$ sudo rm /var/lib/dpkg/lock
```

- **ERROR AL LANZAR LA SIMULACIÓN DEL MODELO IRB120**

Si se siguen los pasos del Capítulo 5, una vez compilado el paquete “abb:experimental” se ejecuta el siguiente comando para comenzar la simulación del modelo IRB120.

```
$ roslaunch abb_irb120_gazebo irb120_3_58_gazebo.launch
```

```
[ INFO] [1568358152.500286198, 0.098000000]: Starting gazebo_ros_control plugin in nam
espace: /
[ INFO] [1568358152.502899940, 0.098000000]: gazebo_ros_control plugin is waiting for
model URDF in parameter [robot_description] on the ROS param server.
[ WARN] [1568358152.622479048, 0.098000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_1'.
[ERROR] [1568358152.624913189, 0.098000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_1
[ WARN] [1568358152.625145791, 0.098000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_2'.
[ERROR] [1568358152.627503527, 0.098000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_2
[ WARN] [1568358152.627587540, 0.098000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_3'.
[ERROR] [1568358152.639773144, 0.098000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_3
[ WARN] [1568358152.639872109, 0.098000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_4'.
[ERROR] [1568358152.642000759, 0.098000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_4
[ WARN] [1568358152.642097469, 0.098000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_5'.
[ERROR] [1568358152.645070702, 0.098000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_5
[ WARN] [1568358152.645168157, 0.098000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_6'.
[ERROR] [1568358152.648439093, 0.098000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_6
[ INFO] [1568358152.660479743, 0.098000000]: Loaded gazebo_ros_control.
```

Figura C.1: Terminal que muestra el error tras ejecutar el código.

ANEXO C

Tal y como se muestra en la Figura C.1, al ejecutar el comando anterior se generan varios errores y avisos que no dejan abrir correctamente el archivo. Es un mismo error, pero duplicado para todos los “joints” y se produce al cargar los controladores de ROS. El problema se debe a que, en la configuración de los controladores, no se especifica el valor de la ganancia en ninguno de ellos.

La cinemática del robot se comienza a programar en el archivo “irb120_3_58_macro.xacro”. A cada “joint” se le asigna una configuración: “Effort Joint Interface”, “Velocity Joint Interface” o “Position Joint Interface”. Cada una de ellas devuelve el valor de la fuerza, velocidad o posición en función de cómo se ha configurado. Como en este caso nos interesa saber la posición del “joint”, se configuran todos con la misma “interface”. Se modifica el “joint_1”, tal y como se visualiza en la Figura C.2, para evitar que aparezca el “warning” de la Figura C.1.

```
<!-- get base ABB IRB120 model -->
<xacro:abb_irb120_3_58 prefix="${prefix}" />

<!-- transmission list -->
<transmission name="${prefix}tran1">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="${prefix}joint_1">
      <hardwareInterface>hardware_interface/PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="${prefix}motor1">
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
<transmission name="${prefix}tran2">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="${prefix}joint_2">
      <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="${prefix}motor2">
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
<transmission name="${prefix}tran3">
  <type>transmission_interface/SimpleTransmission</type>
  <joint name="${prefix}joint_3">
      <hardwareInterface>PositionJointInterface</hardwareInterface>
  </joint>
  <actuator name="${prefix}motor3">
    <mechanicalReduction>1</mechanicalReduction>
  </actuator>
</transmission>
```

Figura C.2: Parte del archivo “irb120_3_58_macro.xacro”.

A continuación, se modifica el archivo “irb120_3_58_arm_controller.yaml.” como muestra la Figura C.3. La única modificación que se realiza es programar una ganancia para el “joint_1”, los datos numéricos son meramente arbitrárias, para comprobar que el error se ha subsanado.

```yaml
arm_controller:
  type: position_controllers/JointTrajectoryController
  joints:
     - joint_1
     - joint_2
     - joint_3
     - joint_4
     - joint_5
     - joint_6
  gains:
    joint_1: {p: 1000.0, i: 500.0, d: 20.0}
  constraints:
      goal_time: 0.6
      stopped_velocity_tolerance: 0.05
      joint_1: {trajectory: 0.1, goal: 0.1}
      joint_2: {trajectory: 0.1, goal: 0.1}
      joint_3: {trajectory: 0.1, goal: 0.1}
      joint_4: {trajectory: 0.1, goal: 0.1}
      joint_5: {trajectory: 0.1, goal: 0.1}
      joint_6: {trajectory: 0.1, goal: 0.1}
  stop_trajectory_duration: 0.5
  state_publish_rate:  25
  action_monitor_rate: 10
```

Figura C.3: Archivo "irb120_3_58_arm_controller.yaml.".

Se vuelve a lanzar la simulación:

```
$ roslaunch abb_irb120_gazebo irb120_3_58_gazebo.launch
```

```
[ INFO] [1568358238.781288421, 0.138000000]: Starting gazebo_ros_control plugin in nam
espace: /
[ INFO] [1568358238.782752292, 0.138000000]: gazebo_ros_control plugin is waiting for
model URDF in parameter [robot_description] on the ROS param server.
[ WARN] [1568358238.900947780, 0.138000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_2'.
[ERROR] [1568358238.903260729, 0.138000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_2
[ WARN] [1568358238.903400703, 0.138000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_3'.
[ERROR] [1568358238.905472954, 0.138000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_3
[ WARN] [1568358238.905632977, 0.138000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_4'.
[ERROR] [1568358238.907670457, 0.138000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_4
[ WARN] [1568358238.907862393, 0.138000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_5'.
[ERROR] [1568358238.910049366, 0.138000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_5
[ WARN] [1568358238.910228390, 0.138000000]: Deprecated syntax, please prepend 'hardwa
re_interface/' to 'PositionJointInterface' within the <hardwareInterface> tag in joint
 'joint_6'.
[ERROR] [1568358238.915278531, 0.138000000]: No p gain specified for pid.  Namespace:
/gazebo_ros_control/pid_gains/joint_6
[ INFO] [1568358238.925939796, 0.138000000]: Loaded gazebo_ros_control.
[ WARN] [1568358238.926199034, 0.139000000]: The default_robot_hw_sim plugin is using
the Joint::SetPosition method without preserving the link velocity.
[ WARN] [1568358238.926238173, 0.139000000]: As a result, gravity will not be simulate
d correctly for your model.
```

Figura C.4: Terminal tras modificar parámetros en el "joint_1".

Como se observa en la Figura C.4, para el “joint_1” ya no aparece ningún error, ni ningún aviso. Por lo tanto, para poder ejecutar la simulación correctamente bastaría con repetir las dos acciones anteriores para los seis “joints”.

D. CÓDIGO PARA TAREA DE MANIPULACIÓN CON PINZA

En este cuarto Anexo se transcribe el programa utilizado para la aplicación presentada en el Capítulo 6.2. La acción consiste en coger y trasladar una lata con el brazo robótico. En el propio programa, mediante comentarios se va explicando para que sirve cada parte de los algoritmos.

```
#!/usr/bin/python
#

from trajectory_msgs.msg import JointTrajectory
from std_msgs.msg import Header
from trajectory_msgs.msg import JointTrajectoryPoint
import rospy
import argparse
import actionlib
import control_msgs.msg
import time

def main():

    #Se inicia el nodo y se llama al topic correspondiente
    rospy.init_node('send_joints')
    pub = rospy.Publisher('/trajectory_controller/command',JointTrajectory,queue_size=10)

    # Se crea el mensaje
    traj = JointTrajectory()
    traj.header = Header()
    traj.joint_names = ['shoulder_pan_joint', 'shoulder_lift_joint','elbow_joint',
                        'wrist_1_join t', 'wrist_2_joint','wrist_3_joint']

    # Bucle para crear el vector de la primera posición del robot
    rate = rospy.Rate(10)
    for x in range(0, 35000):
```

```
    traj.header.stamp = rospy.Time.now()
    pts = JointTrajectoryPoint()
    pts.positions = [1.15, -0.75, 0.85, 0.0, 0.168, 0.1]
    pts.time_from_start = rospy.Duration(1.0)

    # Se publica el mensaje con la trayectoria
    traj.points = []
    traj.points.append(pts)
    pub.publish(traj)

   # Se crea la función para llamar el topic que controla el gripper
   client = actionlib.SimpleActionClient('/gripper_controller/gripper_cmd',
      control_msgs.msg.GripperCommandAction)

   #Se le manda a la pinza el valor de su apertura que oscila entre 0 (totalmente abierta) y
0.8 (cerrada).
   client.wait_for_server()
   goal = control_msgs.msg.GripperCommandGoal()
   goal.command.position = 0.19
   goal.command.max_effort = -1.0
   client.send_goal(goal)

   time.sleep(5) # espera en segundos

   # Segundo movimiento con la lata atrapada en la pinza
   rate = rospy.Rate(10)
   for x in range(0, 50000):

    traj.header.stamp = rospy.Time.now()
    pts = JointTrajectoryPoint()
    pts.positions = [-2, -1, 0.85, 0.0, 0.172, 0.1]
    pts.time_from_start = rospy.Duration(1.0)
    traj.points = []
```

```
        traj.points.append(pts)
        pub.publish(traj)



    # El gripper se abre y la lata queda posicionada sobre la barra de madera
    goal.command.position = 0.1
    goal.command.max_effort = -1.0
    client.send_goal(goal)

    time.sleep(2) # espera en segundos

    # Últimos dos movimientos para llevar al robot a su posición de origen
    for x in range(0, 10000):

        traj.header.stamp = rospy.Time.now()
        pts = JointTrajectoryPoint()
        pts.positions = [-2, -1.2, 1, 0.0, 0.172, 0.09]
        pts.time_from_start = rospy.Duration(1.0)
        traj.points = []
        traj.points.append(pts)
        pub.publish(traj)

    for x in range(0, 50000):

        traj.header.stamp = rospy.Time.now()
        pts = JointTrajectoryPoint()
        pts.positions = [0, -1.8, 1, 0.0, 0.0, 0.0]
        pts.time_from_start = rospy.Duration(1.0)
        traj.points = []
        traj.points.append(pts)
        pub.publish(traj)

if __name__ == '__main__':
    try:
```

```
    main()
#Ejecuta la función definida anteriormente
  except rospy.ROSInterruptException:
    print ("Program interrupted before completion"
```

La parte de movimiento del robot se explica en el apartado 5.3.2 (Figura 5.16). En este archivo para ejecutar varios movimientos se repite el bucle, modificando el vector posición. Los tiempos de duración del bucle se ajustan dependiendo de la duración del movimiento.

En el código se incluye la programación para modificar la apertura de la pinza y poder así atrapar un objeto. Para ello primero se crea una función, donde se llama a la acción mediante el "topic" correspondiente. Esta acción es del tipo "control_msgs"

```
  # Se crea la función para llamar el topic que controla el gripper
  client = actionlib.SimpleActionClient('/gripper_controller/gripper_cmd',
    control_msgs.msg.GripperCommandAction)
```

Posteriormente se hace referencia a la función "client" y se le manda el valor de la apertura correspondiente tanto para cerrar como para abrir la pinza. Junto con el valor de la apertura se escribe un valor para la fuerza de la pinza.

```
  #Se le manda a la pinza el valor de su apertura que oscila entre 0 (totalmente abierta) y
0.8 (cerrada).
  client.wait_for_server()
  goal = control_msgs.msg.GripperCommandGoal()
  goal.command.position = 0.19
  goal.command.max_effort = -1.0
  client.send_goal(goal)
```

E. CÓDIGO PARA TAREA DE MANIPULACIÓN CON "VACUUM GRIPPER"

En este quinto Anexo se transcribe el código utilizado para la aplicación presentada en el Capítulo 6.3. A modo de resumen, el robot se encarga del transportar dos piezas. La primera de ellas la lleva desde la caja Nº7 al AGV situado a su izquierda y la segunda la traslada desde la cinta transportadora hasta el AGV de su derecha. Como en el anterior Anexo, se han ido escribiendo comentarios que detallan cada parte del programa.

```
#!/usr/bin/python
#

from trajectory_msgs.msg import JointTrajectory
from std_msgs.msg import Header
from trajectory_msgs.msg import JointTrajectoryPoint
import rospy
import time
from osrf_gear.srv import VacuumGripperControl
from osrf_gear.srv import AGVControl

#Función para activar el griper o desactivarlo
def control_gripper(enabled):
    gripper_control = rospy.ServiceProxy('/ariac/gripper/control', VacuumGripperControl)
    response = gripper_control(enabled)

#Función para controlar el agv
def control_agv(index, kit_type):
    name = '/ariac/agv' + str(index)
    agv_control = rospy.ServiceProxy(name, AGVControl)
    response = agv_control(kit_type)

#Función principal
def main():
```

```
#Se inicia el nodo
rospy.init_node('send_joints')
pub = rospy.Publisher('/ariac/arm/command',JointTrajectory,queue_size=10)

# Se crea el mensaje y se nombran los joints del robot.
traj = JointTrajectory()
traj.header = Header()
traj.joint_names = ['elbow_joint', 'linear_arm_actuator_joint', 'shoulder_lift_joint',
                    'shoulder_pan_joint', 'wrist_1_joint', 'wrist_2_joint', 'wrist_3_joint']

# Bucle para iniciar el primer movimiento
rate = rospy.Rate(10)
for x in range(0, 50000):
    traj.header.stamp = rospy.Time.now()
    pts = JointTrajectoryPoint()
    pts.positions = [1.85, 0.35, -0.38, 2.76, 3.67, -1.51, 0.0]
    pts.time_from_start = rospy.Duration(1.0)
    # Se envía la trayectoria y se publica el mensaje
    traj.points = []
    traj.points.append(pts)
    pub.publish(traj)

# Se activa el vacuum gripper
success = control_gripper(True)
time.sleep(0.5)

# Segundo y tercer movimiento para acercar el robot al AGV
for x in range(0, 20000):
    pts.positions = [1.76, 0.38, -1.38, 1.5, 3.27, -1.51, 0.0]
    traj.points = []
    traj.points.append(pts)
    pub.publish(traj)
```

```
for x in range(0, 50000):
    pts.positions = [1.76, 2.06, -0.63, 1.5, 3.27, -1.51, 0.0]
    traj.points = []
    traj.points.append(pts)
    pub.publish(traj)

# Se desactiva el gripper para dejar la pieza sobre el AGV
success = control_gripper(False)
time.sleep(0.5)

# Se activa el movimiento del agv Nº1
success = control_agv(1, 'order_0_kit_0')

# El robot se acerca a la cinta transportadora
for x in range(0, 50000):
    traj.header.stamp = rospy.Time.now()
    pts = JointTrajectoryPoint()
    pts.positions = [0, 1.51, -1.13, 3.14, 3.77, -1.51, 0.0]
    pts.time_from_start = rospy.Duration(1.0)

# Se activa el vacuum gripper
success = control_gripper(True)
time.sleep(0.5)

# El robot desplaza la pieza al AGV Nº2
for x in range(0, 20000):
    pts.positions = [-1.76, 0.38, -1.38, 1.5, 3.27, -1.51, 0.0]
    traj.points = []
    traj.points.append(pts)
    pub.publish(traj)

for x in range(0, 50000):
    pts.positions = [-1.76, 2.06, -0.63, 1.5, 3.27, -1.51, 0.0]
    traj.points = []
```

```
        traj.points.append(pts)
        pub.publish(traj)

    # Se desactiva el vacuum gripper
    success = control_gripper(False)
    time.sleep(0.5)
    # Se activa el movimiento del agv Nº2
    success = control_agv(2, 'order_0_kit_0')

    # El robot vuelve a la posición de inicio
    for x in range(0, 50000):
        traj.header.stamp = rospy.Time.now()
        pts = JointTrajectoryPoint()
        pts.positions = [0, 1.51, -1.13, 3.14, 3.77, -1.51, 0.0]
        pts.time_from_start = rospy.Duration(1.0)

if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        print ("Program interrupted before completion")
```

De nuevo en este programa se utiliza la parte movimiento del robot explicada en el apartado 5.3.2 (Figura 5.16). Para ejecutar varios movimientos se repite el bucle, modificando el vector posición. En este caso, al definir el vector posición se escribe un elemento más. Este elemento corresponde a la barra metálica sobre la que se mueve el robot en la simulación (*'linear_arm_actuator_joint'*), y que le permite acercarse al AGV para depositar la pieza.

Se programan dos funciones nuevas, utilizadas para activar o desactivar el "vacuum gripper" y para lanzar la tarea al AGV. La función del gripper es "booleana" ya que solo admite los valores de "True" y "False". Se crea llamando al "topic" correspondiente, que en este caso es *'/ariac/gripper/control'*. Las diferentes funciones del AGV ya vienen programadas en otro archivo (AGV_Control). En este código solo

es necesario crear una función como la que se muestra a continuación, para pasarle que tipo de acción tiene que realizar.

```
#Función para activar el griper o desactivarlo
def control_gripper(enabled):
    gripper_control = rospy.ServiceProxy('/ariac/gripper/control', VacuumGripperControl)
    response = gripper_control(enabled)

#Función para controlar el agv
def control_agv(index, kit_type):
    name = '/ariac/agv' + str(index)
    agv_control = rospy.ServiceProxy(name, AGVControl)
    response = agv_control(kit_type)
```

```
    # Se desactiva el gripper para dejar la pieza sobre el AGV
    success = control_gripper(False)
    time.sleep(0.5)

    # Se activa el movimiento del agv
    success = control_agv(1, 'order_0_kit_0')
```

Para seleccionar el AGV que queremos que realice la acción basta con indicarlo en “*control_agv*” con el número 1 ó 2. Solo hay una orden programada que se utiliza para cualquiera de los AGV y cuyo objetivo es depositar las piezas y volver a la posición de partida vacío.

LISTA DE FIGURAS

2.1 Clasificación de los robots dependiendo de su estructura 4

2.2 Clasificación de los robots industriales en función de sus movimientos 5

2.3 Semejanza entre un brazo humano y un robot industrial de seis ejes 6

2.4 Operaria y cobot comparten espacio de trabajo en la industria 7

3.1 Componentes de ROS. 9

3.2 Ecosistema que rodea a ROS 10

3.3 Estructura básica de un paquete ROS 11

3.4 Sencillo esquema del sistema de funcionamiento entre nodos. 12

3.5 Simulación de un dron en Gazebo. 13

3.6 Visualización de un robot en RViz. 14

3.7 Dos esquemas de los distintos componentes de un fichero URDF. 15

3.8 Estructura de comunicación del nodo “move_group”. 16

4.1 Captura de pantalla de la terminal después de haber realizado los tres primeros pasos. 18

4.2 Recorte de las últimas líneas del código del archivo “.bashrc”. 21

4.3 Información de salida al ejecutar el comando “roscore”. 22

4.4 Demonstración de la segunda prueba de funcionamiento. 23

4.5 Paquetes necesarios para llevar a cabo la simulación del “Turtlebot3”. 24

4.6 Terminal donde se controlan los movimientos de la simulación del modelo “waffle” en Gazebo. 25

4.7 Vista del entorno RViz al activar el sensor de la cámara. 26

4.8 Simulación del modelo “Burger” en Gazebo. 27

5.1 Simulación de un robot ABB en el entorno de RobotStudio. 28

5.2 Contenido del paquete “abb_experimental” tras su compilación, destacando las dos carpetas con las que se va a trabajar. 29

5.3 Simulación del modelo IRB120 en Gazebo, mostrando todos los “topics” en activo 30

5.4 Asistente de “MoveIt” tras elegir el archivo para crear su nueva configuración. 31

5.5 Captura del asistente una vez generada la matriz de colisión. 32

5.6 Pestaña de “Virtual Joint”. 32

5.7 Opciones escogidas en el apartado “Planning Groups”. 33

5.8 Grupo “arm” creado con sus respectivos “joints”. 33

5.9 Creación de una posición aleatoria del robot. 34

5.10 Pestaña “ROS Control” para definir un controlador al robot. 35
5.11 Último paso del asistente al generar el paquete de configuración. 35
5.12 Terminales donde se han ejecutado los comandos necesarios para visualizar los “topics” y conocer el tipo de uno en concreto. 36
5.13 Terminal con el comando que se utiliza para mover al robot. 37
5.14 Simulación en Gazebo del robot en su posición de partida. 37
5.15 Simulación del robot una vez se ha ejecutado el comando de la Figura 5.13. 37
5.16 Captura del código escrito en “python”. 38
5.17 Contenido del paquete creado con el asistente de “MoveIt”. 39
5.18 Recorte de la opción “Motion Planning” 40
5.19 Opción “Displays” de la simulación en RViz. 40
5.20 Visualización de RViz tras mover el robot con el marcador interactivo. 41
5.21 Captura de las simulaciones de Gazebo (izquierda) y RViz (derecha). 41
5.22: Recorte del archivo “irb120_3_58_gazebo.launch”. 42
5.23: Parámetros de cada robot al lanzar el archivo “irb120_3_58_gazebo.launch”. 43
5.24: Simulación en Gazebo de dos robots tras haber ejecutado dos trayectorias diferentes. 44
6.1 Proporción de tamaños entre cada uno de los tres modelos de Universal Robots con sus respectivas características. 46
6.2: “Tablet” usada para programar cobots de Universal Robots. 47
6.3: Simulación en Gazebo del modelo UR5 y la pinza robótica 2F-85. 48
6.4: Recorte del archivo “ur5_setup.world”. 49
6.5: Simulación al crear el entorno y antes de ejecutar la tarea. 50
6.6: Imagen del robot durante su acción de llevar la lata sobre la barra de madera. 51
6.7: Fragmento del archivo “escenario_1.yaml”. 53
6.8: Escenario previo a la realización de la tarea de manipulación.. 54
6.9: Robot desplazándose hasta el AGV Nº1 55